MASTER IN COMPUTER SCIENCE

Workflows et scénarios d'exécution de services dans un GRID sémantique

Mainil, Rémy

Award date:
2006

Facultés Universitaires Notre-Dame de la Paix. Namur

Institut d'Informatique

Année Académique 2005 - 2006

Workflows et scénarios d'exécution de services dans un GRID sémantique

Rémy Mainil

Mémoire présenté en vue de l'obtention du grade de Licencié en Informatique.

Résumé

Le projet BIGRE s'inscrit dans le cadre de la bioinformatique et est issu de la volonté de faciliter l'utilisation des ressources bioinformatiques aux utilisateurs potentiels. Faciliter l'utilisation des ressources bioinformatiques disponibles nécessite notamment de s'attaquer au problème de l'hétérogénéité de ces différentes ressources. La plateforme BIGRE propose une solution à ce problème en fournissant une sémantique commune pour les ressources qu'elle propose, que ces dernières soient des services spécialement développés pour BIGRE ou des services existants encapsulés dans des services façades.

Dans ce mémoire, nous proposons une architecture capable, sur base de la sémantique commune mise en place et de la description d'un service, de générer de façon dynamique une interface graphique permettant d'exécuter ce service mais également d'en contrôler l'exécution grâce à un moteur de workflows qui, sur base du scénario d'exécution du service, construit les différents éléments de l'interface et guide l'utilisateur dans les différentes étapes du service.

Nous proposons également un prototype afin de valider cette architecture.

Abstract

The BIGRE project lies within the scope of bioinformatic and results from the will to facilitate the use of the bioinformatic resources to their potential users. Facilitating the use of available bioinformatic resources requires to attack the problem of the heterogeneity of these various resources. BIGRE proposes a solution for this problem by providing a common semantics for the resources which are proposed by the platform, which these last are services especially developed for BIGRE or existing services encapsulated in facade services.

In this master thesis, we propose an architecture able, on the basis of the common semantics and the description of a service, to generate dynamically a graphic user interface allowing to carry out this service but also to supervise the use of this service thanks to a workflows engine which, on the basis of the scenario of execution of the service, builts the various elements of the user interface and guides the user in the various stages of the service.

We also propose a prototype in order to validate this architecture.

Remerciements :

A Sophie qui n'a jamais cessé de croire en moi.

A Monsieur Englebert pour son aide.

A ma famille et mes amis pour leur soutien et leurs encouragements.

A Pierre Buyle et Quentin Dallons pour leurs précieuses informations.

Table des matières

Table des figures

Introduction

Bioinformatics is nothing but good, sound, regular biology appropriately dressed so that it can fit into a computer. [CN03, p 1]

La bioinformatique consiste en une discipline issue de la convergence entre la biologie et l'informatique. Plus simplement, il s'agit de l'utilisation de l'informatique pour analyser, stocker, traiter, étudier, etc. les données biologiques issues, par exemple, du génome humain, dans le but de produire des connaissances nouvelles dans le domaine. Apparue dans les années 80, le développement de la bioinformatique a fortement suivi celui des technologies de l'information. Grâce au développement d'Internet, de nombreuses ressources bioinformatiques devinrent disponibles à travers le monde. Parallèlement à cela, un problème fit son apparition. De nombreuses ressources furent développées pour répondre aux besoins précis de certains utilisateurs à un moment donné mais aucun standard n'existait pour « encadrer » ces développements. Il en résulte une forte hétérogénéités des ressources informatiques tant au niveau des formats de données qu'au niveau de l'utilisation des outils. A l'heure actuelle, cette hétérogénéité constitue un frein au développement de la bioinformatique. En effet, il n'est pas envisageable de réécrire et de standardiser toutes les ressources existantes. Il est donc nécessaire de concevoir un moyen d'intégration des différentes ressources disponibles afin de permettre aux utilisateurs d'utiliser de façon optimale les ressources qui sont mises à leur disposition en leur offrant un logiciel capable d'homogénéiser leur utilisation. De telles initiatives sont en cours de développement. Parmi elles, on peut citer BIGRE [1], INFOBIOGEN[2], Taverna [3], BioSide[4], etc. Ce mémoire s'inscrit dans le cadre du projet BIGRE ([BD03]) et propose la construction d'un moteur de workflows pour l'exécution de services bioinformatiques dans BIGRE. La suite de cette introduction présente les différents concepts utiles

[1] `http://www.info.fundp.ac.be/~bigre`

[2] `http://www.infobiogen.fr`

[3] `http://taverna.sourceforge.net`

[4] `http://departements.enst-bretagne.fr/lussi/article.php3?id_article=55`

aux lecteurs pour cerner le cadre dans lequel s'inscrit ce document. Le premier chapitre fournit une introduction à la bioinformatique et présente de façon plus précise les enjeux et problèmes de cette discipline tandis que le deuxième chapitre introduit le concept de workflow. Dans la première partie de ce mémoire sont présentés les différents projets d'intégration de services bioinformatiques. Le troisième chapitre présente le logiciel Taverna, un programme destiné à faciliter l'utilisation et la conception de workflows dans le domaine des Sciences électroniques. Dans le quatrième chapitre, le système Babel, offert par Infobiogen, est présenté. Il s'agit d'un environnement Web intégré qui propose un accès centralisé aux ressources bioinformatiques du domaine publique. Le chapitre suivant est consacré à la plateforme BIGRE, une plateforme d'intégration de services bioinformatiques.

La seconde partie de ce mémoire est consacrée à la construction du moteur de workflows pour l'exécution des services bioinformatiques à travers BIGRE. Y sont présentés, dans les différents chapitres, l'analyse réalisée, la conception d'un prototype et un exemple illustrant l'utilisation de ce dernier.

Pour terminer, un chapitre sera consacré à la conclusion de ce travail en expliquant les difficultés rencontrées et en apportant les prolongements envisageables.

Chapitre 1

La Bioinformatique

Ce chapitre constitue une synthèse des informations concernant la bioinformatique recueillies dans les références suivantes : [Inf, Den04, BD03, CN03].

La bioinformatique est une branche de la biologie qui fait son apparition dans les années 80, en plein essor de la génomique, avec les premières banques de biomolécules (EMBL [1] et GenBank [2]). Elle est issue d'une approche interdisciplinaire basée sur la biologie, les mathématiques et l'informatique. Dans un premier temps, l'informatique était utilisée pour stocker les quantités de plus en plus grandes d'informations issues de la génomique. Cependant, avec le temps, les capacités de gestion, de calcul et d'automatisation des opérations répétitives ont conduit à la mise en place d'une nouvelle méthode de travail. La bioinformatique est dès lors devenue une branche à part entière de la biologie, baptisée approche « in silico », au même titre que les approches classiques « in situ » (sur site), « in vivo » (sur être vivant) et « in vitro » (en éprouvette). Cette nouvelle approche, contrairement aux approches par expérience « aveugle » tente, via trois méthodes complémentaires, de faire émerger des concepts biologiques originaux qui devront être testés et validés ultérieurement par expérimentation. Les trois méthodes principales de la bioinformatique sont les suivantes :

1. Méthode comparative : des logiciels explorent les bases de données et recherchent dans les gènes et protéines déjà identifiés et annotés par d'autres équipes scientifiques des similitudes ou rapprochements avec des séquences ou structure inconnues et que l'on cherche à identifier.
2. Méthode statistique : des logiciels appliquent des analyses statistiques aux données pour tenter d'en dégager des règles et des contraintes pré-

[1] `http://www.ebi.ac.uk/embl/index.html`
[2] `http://www.ncbi.nlm.nik.gov/GenBank/`

sentant un caractère systématique, régulier ou général.

3. Approche par modélisation : approche probabiliste, elle consiste à étudier les objets (séquences, structures, ...) à travers la construction d'un modèle qui tente d'en extraire les propriétés communes. La relation entre les objets d'étude (et / ou leur reconnaissance) est alors exprimée en référence à ce modèle optimal commun.

Le rôle de la bioinformatique a donc évolué pour passer d'un rôle de gestion de données à un rôle d'exploration, d'analyse de l'information génomique et génétique stockée pour produire de nouvelles connaissance dans le domaine. Ses axes privilégiés sont :

- la formalisation de l'information génétique.
- l'analyse des séquences (biomolécules) et de leur structure.
- l'interprétation biologique de l'information génétique.
- l'intégration des données (réseaux d'interactions génétiques, protéiques, ...).
- la prédiction fonctionnelle.

Ses domaines d'application, quant à eux, sont aussi nombreux que diversifiés. En voici quelques exemples :

- Médecine
  - Diagnostic de maladies génétiques
  - Mise au point de nouveaux médicaments
- Agro-alimentaire
  - Rendement et qualité
  - Résistance aux pesticides, aux pathogènes, ...
- Environnement
  - Lutte contre les pollutions chimiques ou biologiques
  - Nouvelle sources d'énergie

Pour terminer, il serait difficile de parler de la bioinformatique sans aborder les réseaux. Ces derniers ont nécessité la mise en place de nouveaux protocoles de communication ainsi que la restructuration des groupes de travail mais rendent disponibles de nombreuses ressources à travers le monde. La bioinformatique n'a plus comme frontières que la disparité et l'hétérogénéité des différentes ressources disponibles. En effet, si de nombreuses ressources bioinformatiques, qu'il s'agissent de bases de données ou d'outils logiciels, sont disponibles sur internet, il s'avère parfois difficile de les trouver ou même de les utiliser tant il peut parfois y avoir des incompatibilités entre les formats de données.

La bioinformatique est donc une discipline assez jeune mais en forte évolution, notamment grâce aux progrès accomplis en informatique. La mouvance actuelle est à l'élaboration de plateformes d'intégration de services bioinfor-

matiques afin de remédier aux problèmes cités ci-dessus. Des projets tels que BIGRE, Infobiogen, Taverna, Bioside, ...ont pour ambition de faciliter la vie des bioinformaticiens en proposant des plateformes à travers lesquels il est possible de rechercher, utiliser, grouper, divers ressources bioinformatiques.

Chapitre 2

Les Workflows

En 1996, The Workflow Management Coalition[Coa] a publié la définition suivante pour un workflow :

> *The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules.*

On y apprend qu'un workflow peut être vu comme l'automatisation d'un processus métier, entièrement ou en partie, durant laquelle des documents, de l'information, des tâches sont passées d'un participant à un autre pour agir selon un ensemble de règles procédurales. En d'autres mots, le terme « workflow » littéralement traduit par « flux de travail », désigne la modélisation d'un ensemble de tâches à accomplir et d'acteurs impliqués dans la réalisation d'un processus métier. Un processus métier étant un ensemble d'interactions, sous forme d'échange d'informations, entre divers acteurs tels que des humains, des applications, des services, ou encore des processus tiers. Le workflow représente donc le chemin selon lequel va transiter l'information entre les différents acteurs pour arriver au résultat souhaité et fournit, entre autre, à chaque acteur les informations nécessaires à la réalisation de sa tâche. Bien que cela ne soit pas exhaustif, on peut distinguer deux grandes familles de workflows : les workflows « scientifiques »et les workflows « industriels ». Les workflows scientifiques se concentrent sur le passage de données à travers différents algorithmes, logiciels ou services alors que les workflows industriels sont plus orientés vers la planification de tâches, les dépendences entre ces tâches ne concernant pas spécialement les informations qui transitent mais, par exemple, la disponibilité de ressources ou des agents humains. Les workflows industriels sont plus génériques que les workflows scientifiques et permettent de représenter n'importe quelle structuration de tâche, que ce soit

dans un logiciel conçu pour un serveur de tâches ou dans le cheminement d'un document dans une entreprise. Les workflows scientifiques, quant à eux, sont principalement utilisés dans des domaines tels que la bioinformatique ou la cheminformatique (équivalent pour la chimie de la bioinformatique) où ils répondent aux besoins d'interconnexion de nombreux outils et de manipulation de nombreux formats de données et de grandes quantités de données. Une fois défini, il reste à suivre le déroulement du workflow. Cela se fait, en général, grâce à un moteur de workflows. Un moteur de workflows est un environnement d'exécution (souvent, un logiciel) qui coordonne les différents processus et activités définis dans le workflow. Dans le contexte d'un workflow définissant le parcours d'un document électronique au sein d'une entreprise, cela permet par exemple de faire suivre automatiquement le document à l'étape suivante une fois que la tâche courante est terminée (ex : une fois que le journaliste à terminé son article, il le signale et l'article est envoyé à un relecteur qui en est alors averti). Dans le contexte d'un workflow scientifique, le moteur de workflows pourrait prendre les données en entrée puis leur appliquer une succession de traitements et d'algorithmes avant de fournir le résultat en sortie.

Première partie

Etat de l'art

Chapitre 3

Taverna

3.1 Introduction

Le projet Taverna fait partie du projet myGrid[1] fondé par l'EPSRC[2] et a pour but de « *fournir un langage et des outils logiciels pour faciliter l'utilisation et la création de workflows dans le domaine des sciences électroniques*[3] ». Distribué sous licence LGPL [4], il s'agit d'un logiciel libre issu d'une collaboration entre l'Institut Européen de Bioinformatique [5], IT Innovation [6], et différentes universités que sont l'université de Newcastle [7], l'université de Manchester [8] et l'université de Nottingham [9]. L'objectif de ce projet est « *de permettre à un biologiste ou un bioinformaticien avec une connaissance limitée en informatique d'effectuer des analyses complexes avec des ressources privées et publiques depuis un simple PC* ». Taverna permet la construction de workflows (principalement en bioinformatique) permettant de réaliser des expériences « in silico » complexes en offrant un moyen d'accès centralisé à de nombreuses ressources bioninformatiques distribuées de par le monde. Dans la section suivante, nous présentons, à travers un exemple, l'utilisation de Taverna, vient ensuite, en guise de conclusion, une section reprenant les points faibles et avantages de ce logiciel.

---

[1] `http://www.mygrid.info`

[2] Engineering and Physical Sciences Research Council (`http://www.epsrc.ac.uk`

[3] [Tava]

[4] Lesser General Public Licence. Voir `http://www.opensource.org/licenses/lgpl-license.php`

[5] `http://www.ebi.ac.uk`

[6] `http://www.it-innovation.soton.ac.uk`

[7] `http://www.cs.ncl.ac.uk`

[8] `http://www.cs.man.ac.uk`

[9] `http://www.mrl.nott.ac.uk`

3.2 Fonctionnement

Les exemples présentés ci-dessous sont issus de la référence suivante : [Tavb]. Le premier exemple montre comment chercher et invoquer un service bioinformatique dans Taverna et le second exemple montre comment créer un worflow. Ces deux exemples ne font qu'illustrer les fonctions premières de Taverna et sont présentés ici à titre informatif. De nombreuses options sont disponibles dans Taverna et seule une faible partie d'entre-elles sont illustrées ci-après.

Lorsqu'on exécute Taverna, une interface multifenêtrée s'ouvre et présente trois fenêtres : un explorateur de modèles, une fenêtre présentant les différents services disponibles et une fenêtre intitulée « Workflow diagram » et qui contient une « page » blanche. La figure 3.1 montre l'interface initiale du logiciel.

Voyons maintenant comment chercher et invoquer un service.

3.2.1 Chercher et invoquer un service

Chercher un service se fait depuis la fenêtre présentant les services disponibles (« Available services », voir figure 3.2). Une arborescence y présente les services mis à la disposition de l'utilisateur. Pour accéder à un service, ce dernier peut procéder de deux manières. Soit il parcourt l'arborescence jusqu'à trouver le service qu'il cherche – ce qui peut se révéler fastidieux vu la quantité de services proposés – soit il utilise le mécanisme de recherche rapide accessible en haut de la liste. Pour ce faire, l'utilisateur entre un ou plusieurs mots-clés dans un champ prévu à cet effet et situé en haut de la liste des services disponibles (voir figure 3.2), puis lance la recherche.

Une fois la recherche lancée, l'arborescence se réduit automatiquement et ne présente plus que les services qui correspondent, lesquels sont écrits en rouge. La figure 3.3 illustre le résultat d'une recherche effectuée sur « GODB-GetNameById ».

Il nous reste maintenant à utiliser le service que nous avons choisi. Pour cela, il suffit de cliquer avec le bouton droit sur le service que l'on veut invoquer. Nous pouvons alors choisir l'option « Invoke » comme cela est illustré à la figure 3.4.

Une fois l'invocation demandée, une nouvelle fenêtre apparaît (voir la figure 3.5). Elle est intitulée « Run Workflow » et c'est dans cette fenêtre que nous allons fournir les données nécessaires pour l'exécution du service.

Sur la gauche de la fenêtre, une arborescence présente la liste des inputs du service. Lorsqu'on sélectionne un de ces inputs, un panneau apparaît sur la droite dans lequel sont présentés des informations sur l'input (si elles sont

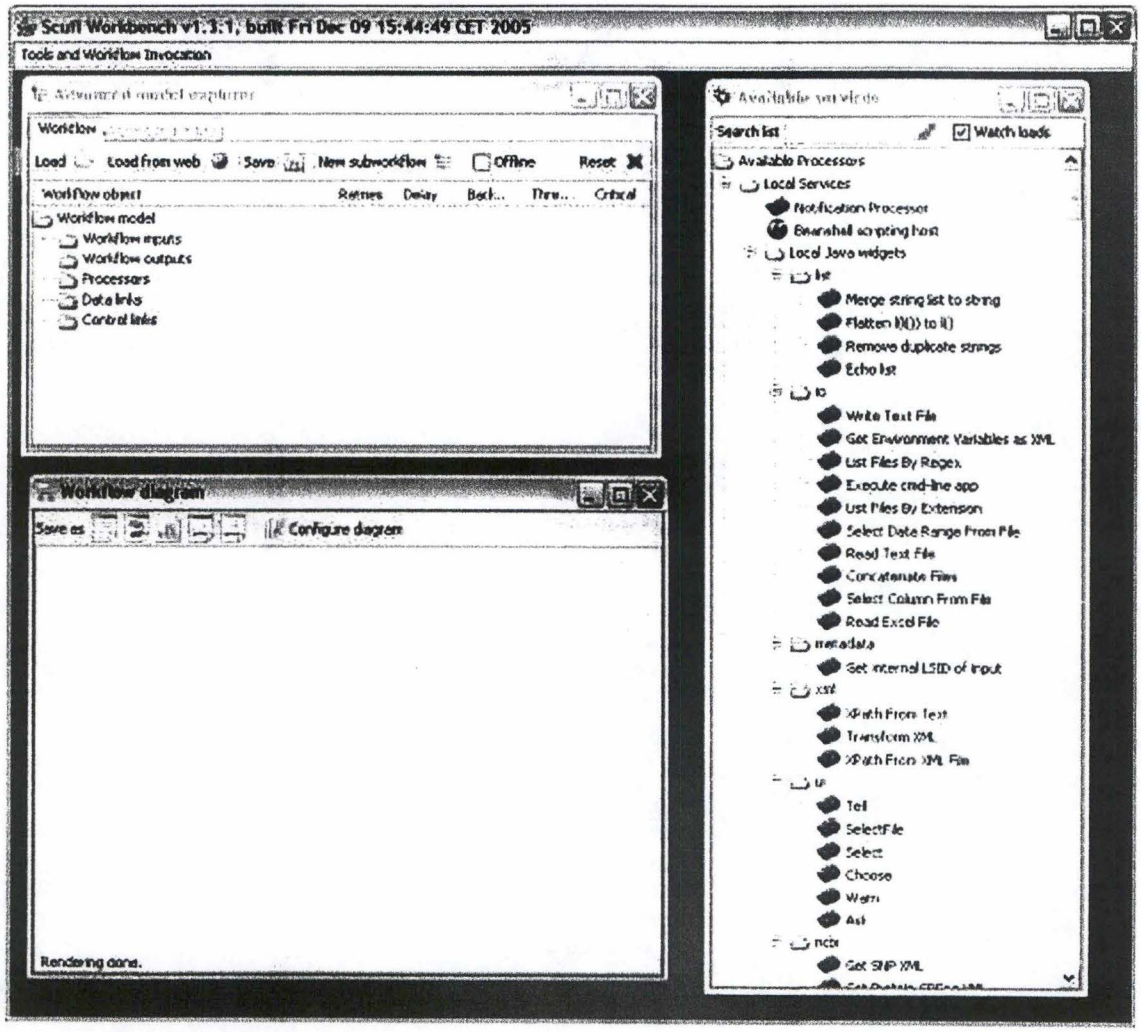

FIG. 3.1 – Interface au démarrage de Taverna

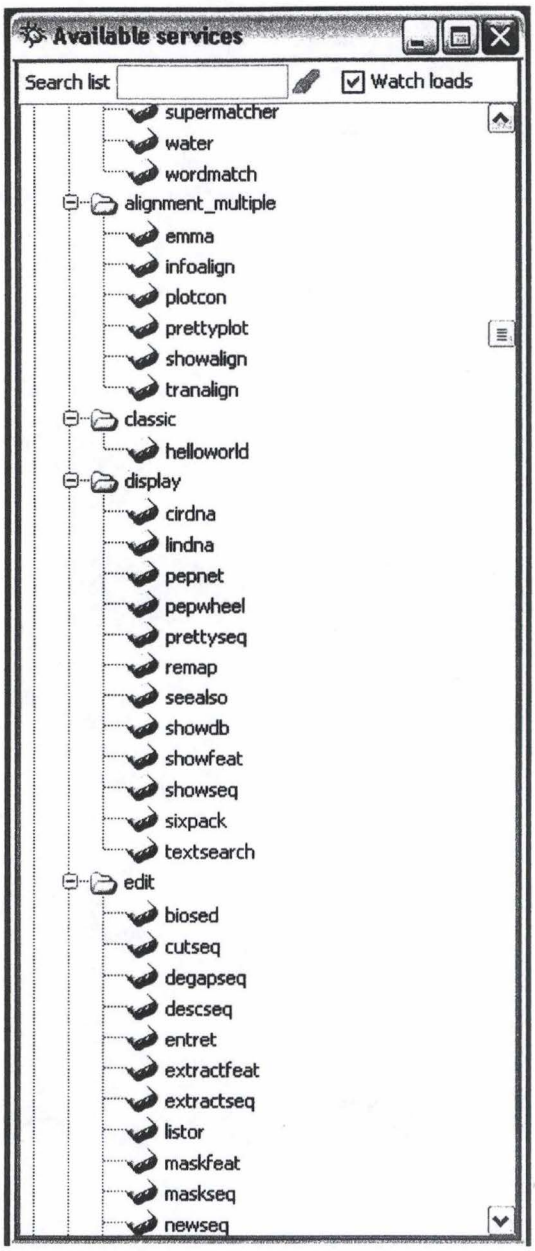

FIG. 3.2 – Arborescence des services disponibles

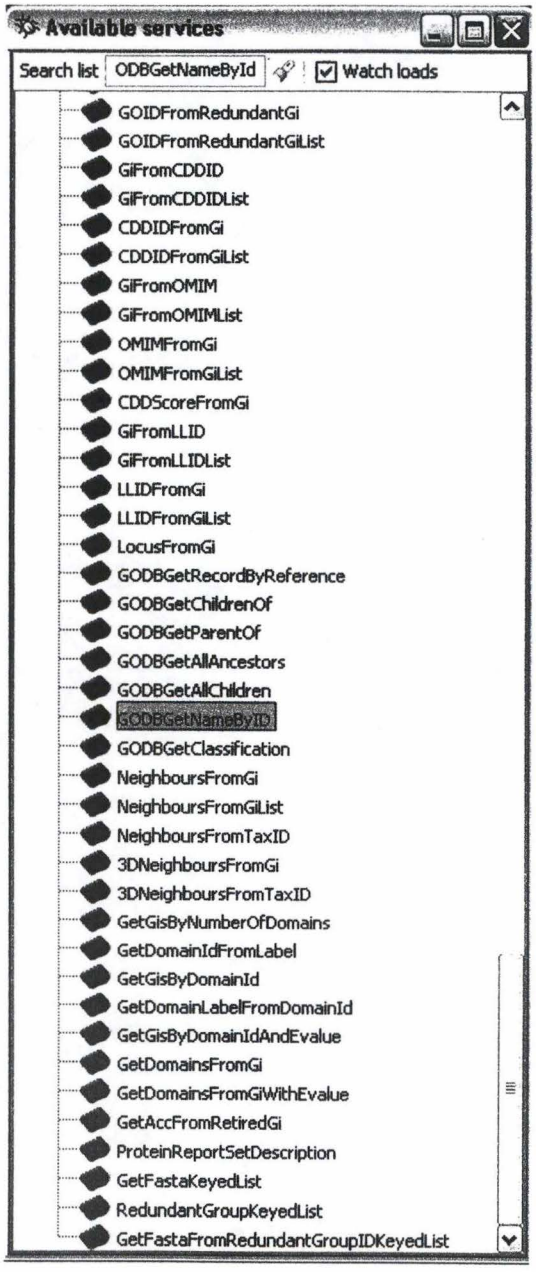

FIG. 3.3 – Résultat de la recherche sur GODBGetNameById

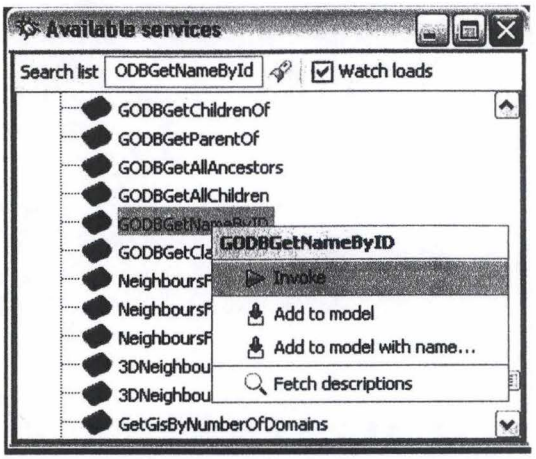

FIG. 3.4 – Invocation d'un service

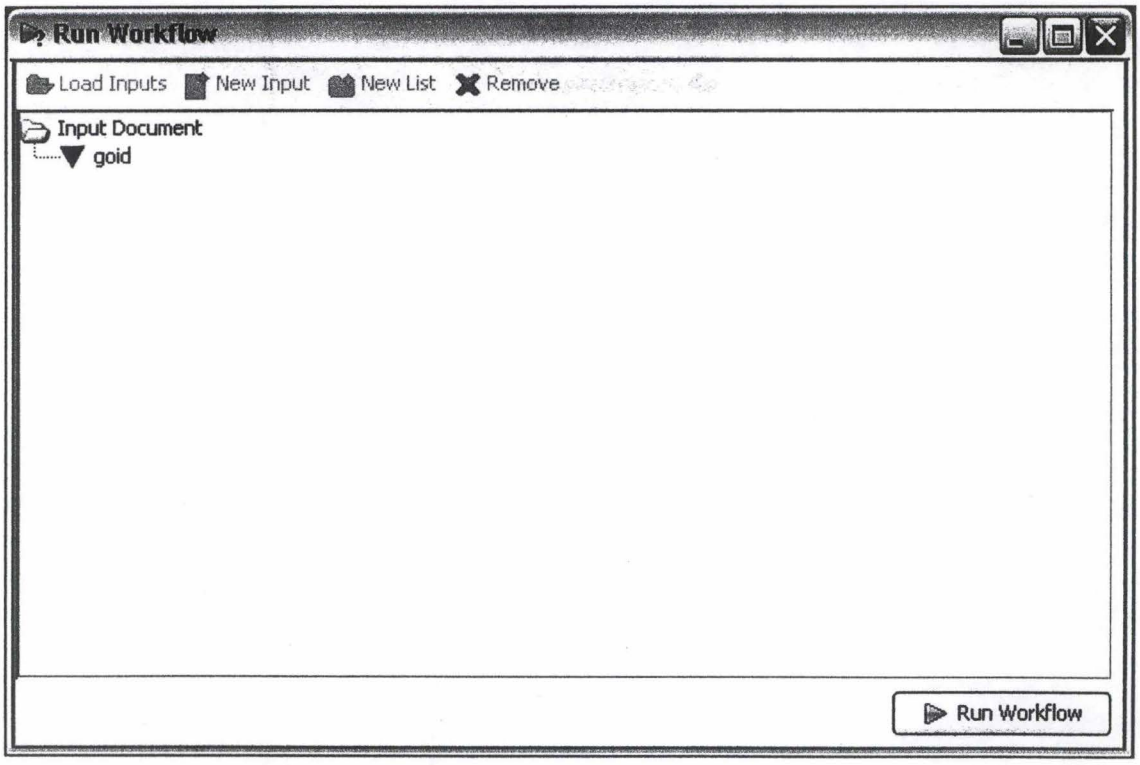

FIG. 3.5 – Fenêtre du service invoqué

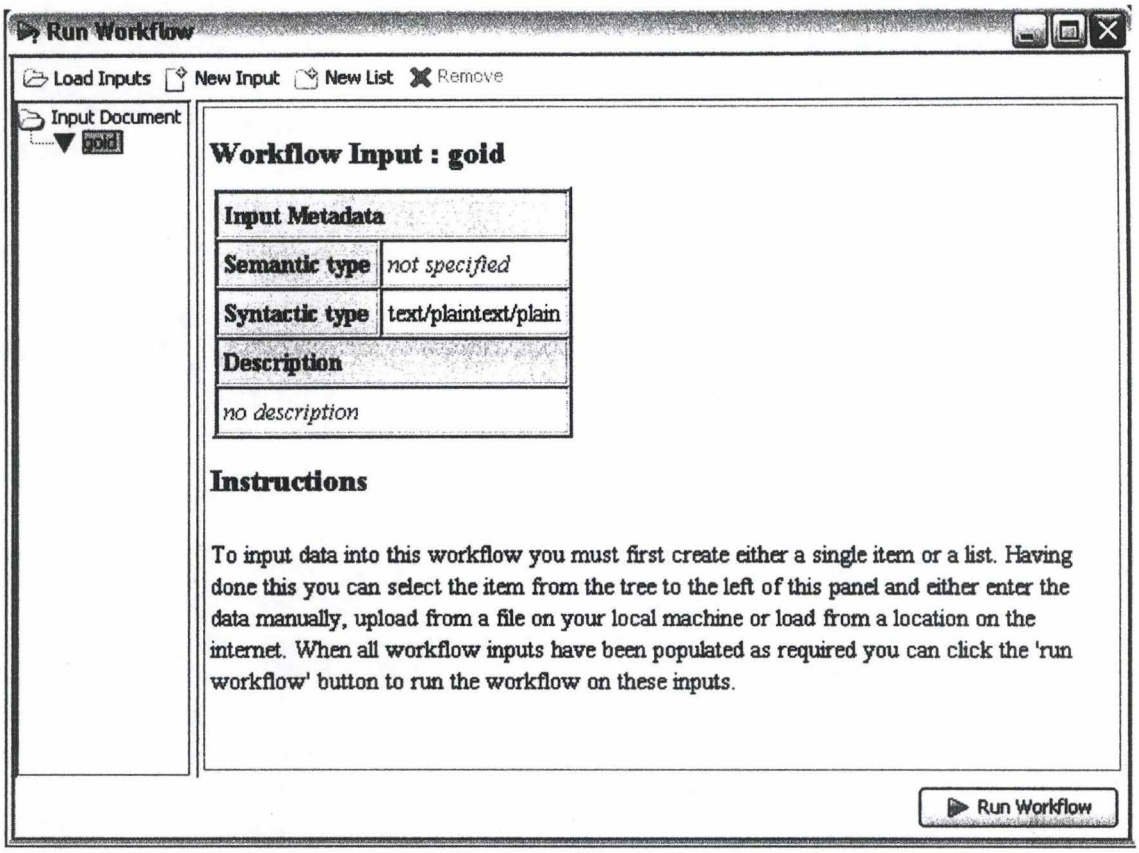

FIG. 3.6 – Description d'un input

disponibles), ainsi que la méthode pour spécifier une valeur pour cet input, comme le montre la figure 3.6.

Pour fournir un input, il suffit alors de cliquer sur le bouton « New Input ». Cela crée un « enfant » à l'input, pour lequel il est possible de spécifier une valeur, comme illustré à la figure 3.7.

Une fois les inputs spécifiés, on peut exécuter le service. Une fenêtre supplémentaire apparait dans laquelle nous pouvons consulter le statut d'exécution du service (voir figure 3.8) ainsi que le graphique représentant le service exécuté. Une fois l'exécution terminée, il devient alors possible de consulter les résultats en cliquant sur l'onglet « Results » (voir figure 3.9).

De là, Taverna offre divers moyens pour éventuellement sauvegarder les données traitées.

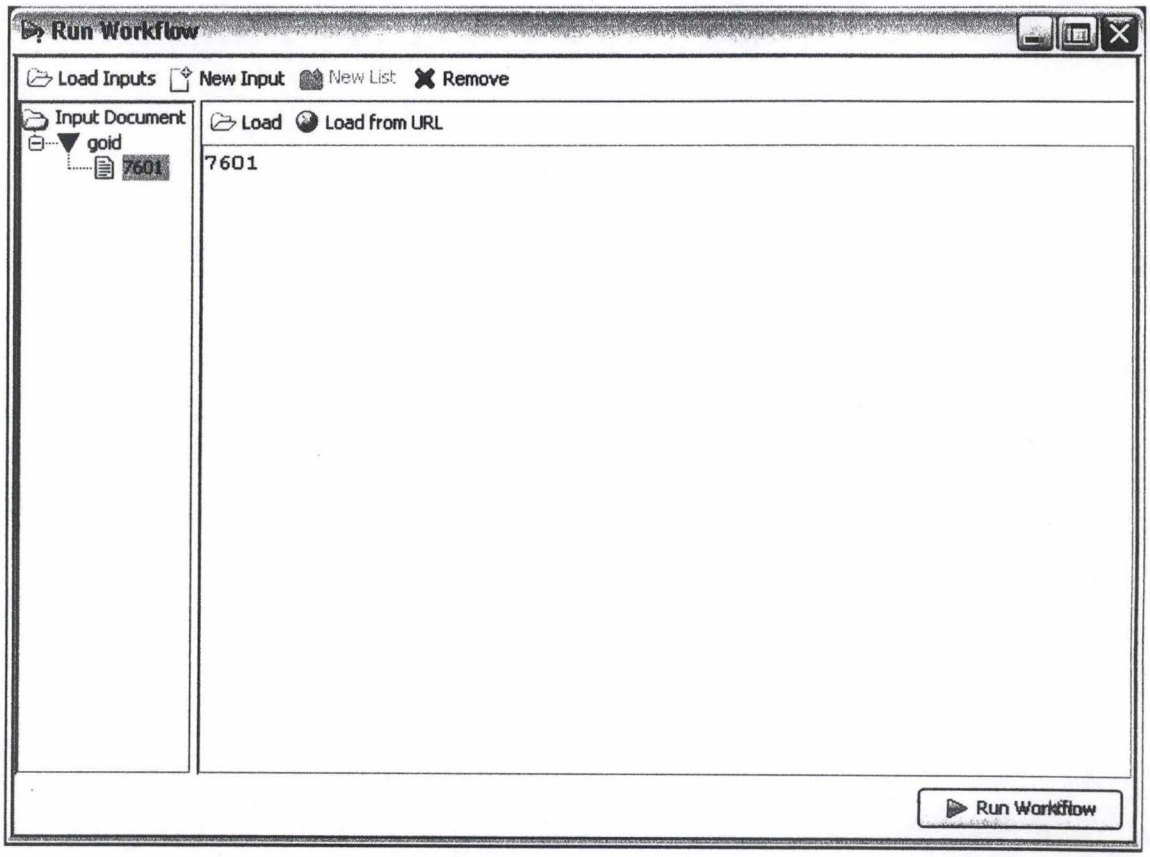

FIG. 3.7 – Ajout d'une valeur pour l'input

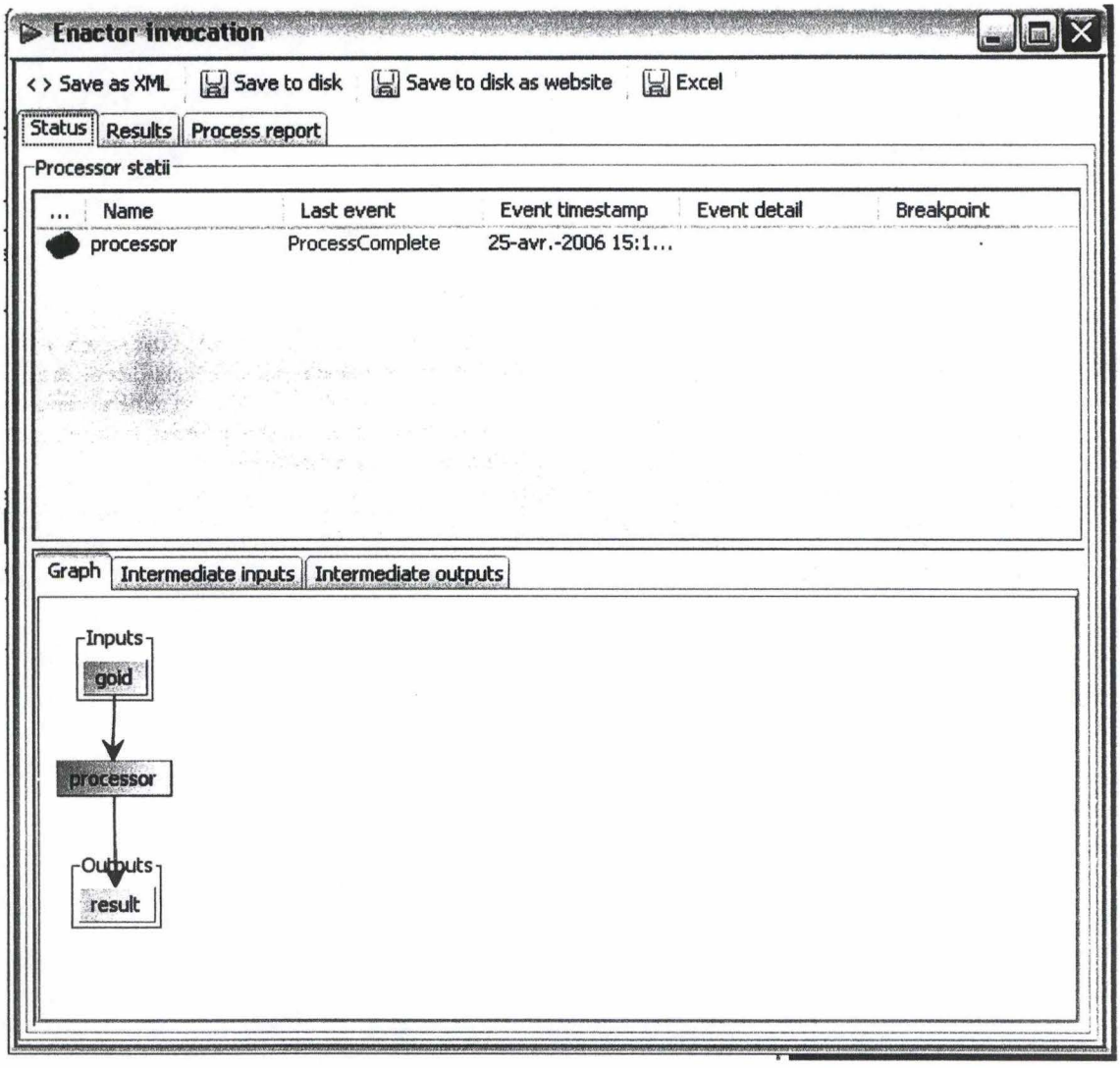

FIG. 3.8 – Statut de l'exécution du service

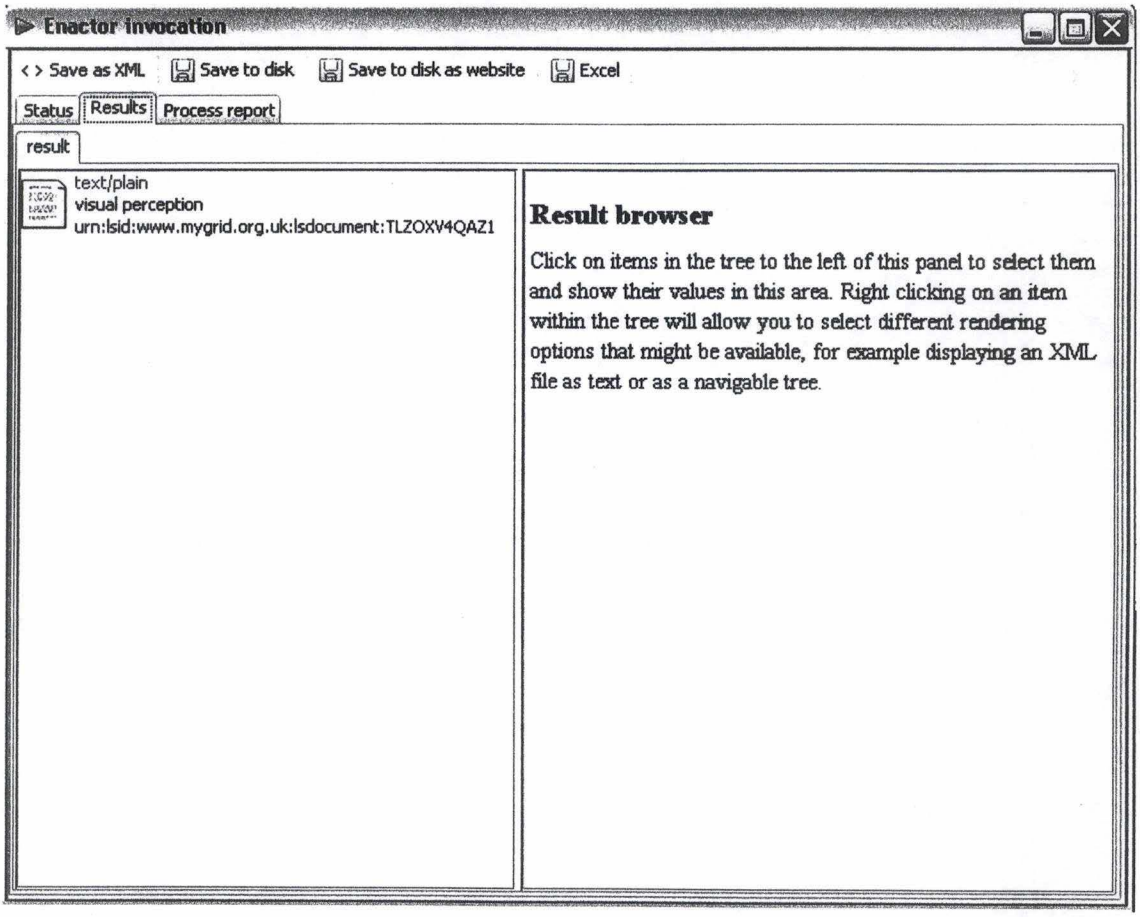

FIG. 3.9 – Résultat de l'exécution du service

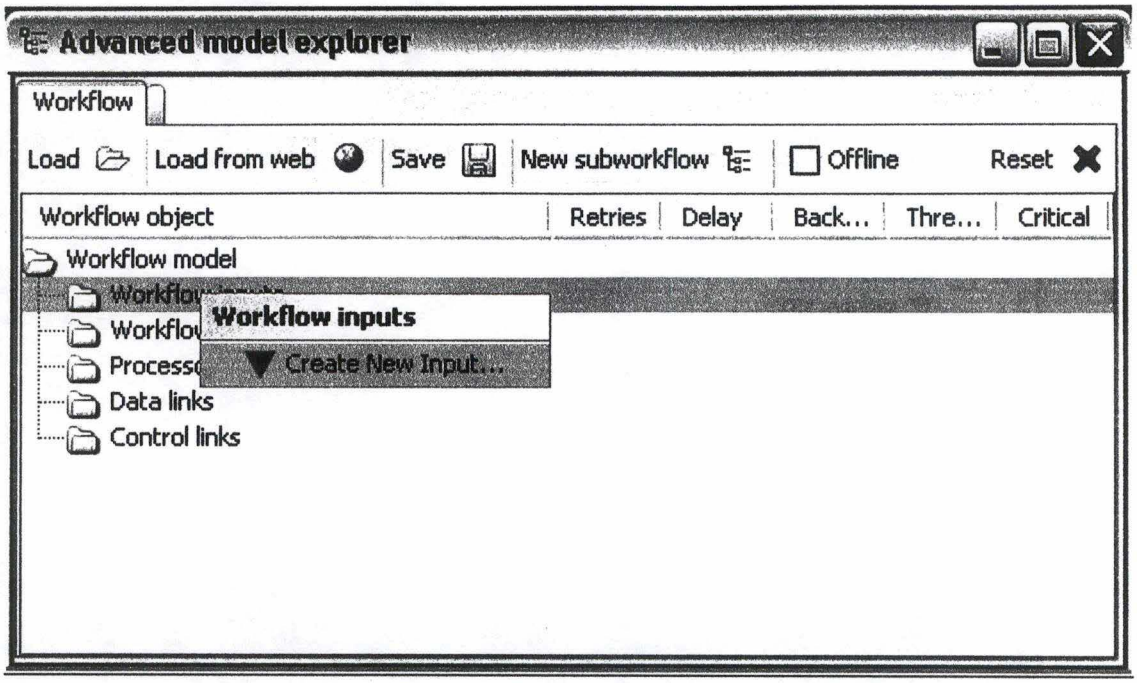

FIG. 3.10 – Création d'un workflow - Ajout des inputs/outputs

3.2.2 Créer un workflow

Outre le fait de permettre l'exécution d'un service bioinformatique, Taverna offre également un moyen de concevoir des workflows dans lesquels il est possible d'enchaîner plusieurs services. Nous allons ici aborder la construction d'un workflow destiné à récupérer une séquence en fonction de son identifiant. La première étape, illustrée à la figure 3.10, consiste à créer un input et un ouput pour le workflow. Une fois ceux-ci créés et nommés, il reste à ajouter au workflow le ou les différents services que l'on veut enchaîner. Pour cela, on peut choisir les services en effectuant une recherche comme cela a été expliqué au point 3.2.1 puis les ajouter au workflow en faisant un glisser/déposer vers l'explorateur de modèles. Au fur et à mesure que l'on ajoute des éléments au workflow, une représentation graphique des éléments est présentée dans la fenêtre intitulée « Workflow Diagram » (voir la figure 3.11). Une fois tous les éléments ajoutés, il reste à les « lier » entre-eux de façon à construire le workflow proprement dit. Dans Taverna, le principe est de chaque fois partir de la source, et de la relier à sa destination. La figure 3.12 illustre la création d'un lien entre la sortie d'un service et la sortie du workflow. Losque tous les liens ont été créés (voir la figure 3.13), il ne reste plus qu'à exécuter le workflow comme cela a été décrit au point 3.2.1.

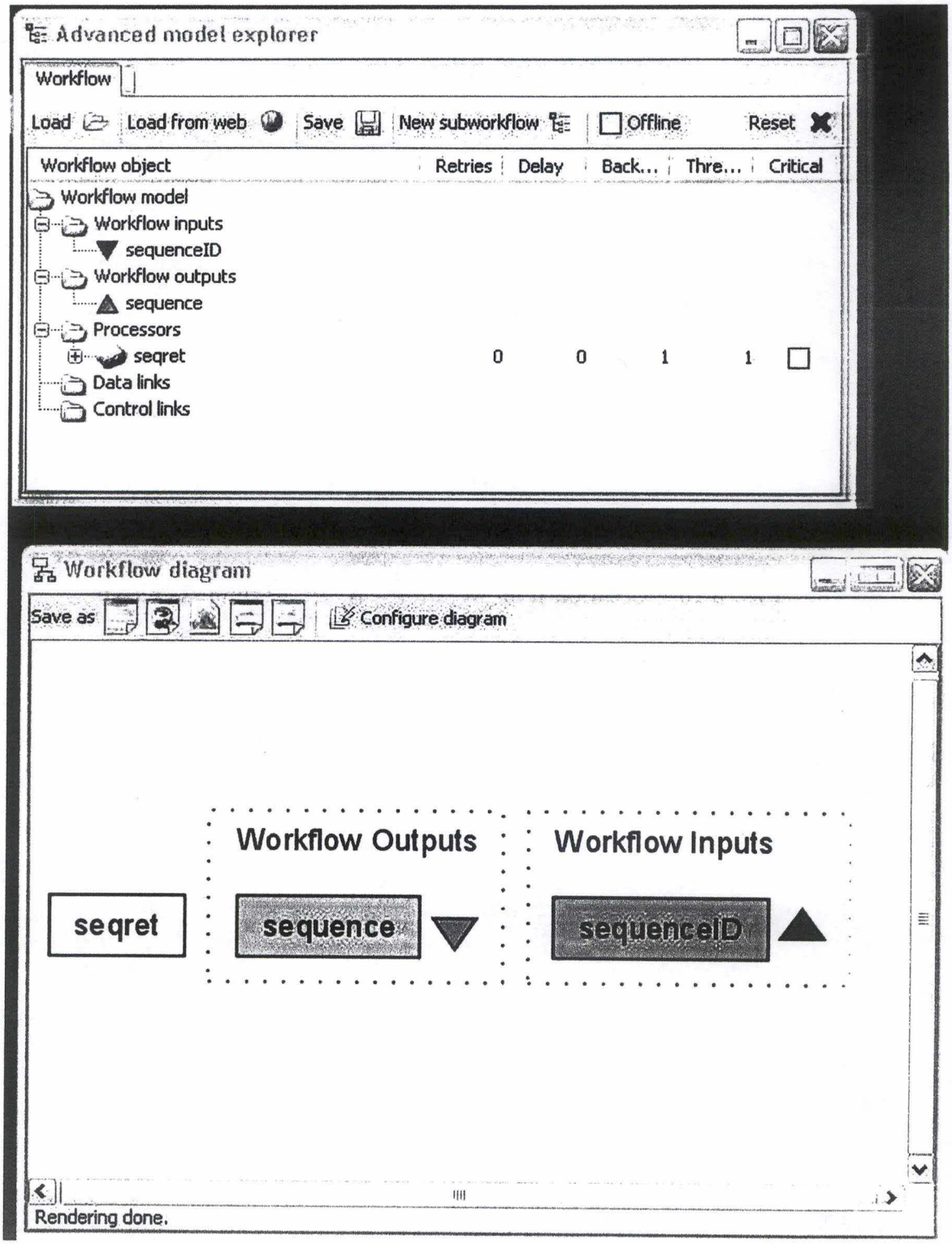

FIG. 3.11 – Création d'un workflow - Ajout d'un service

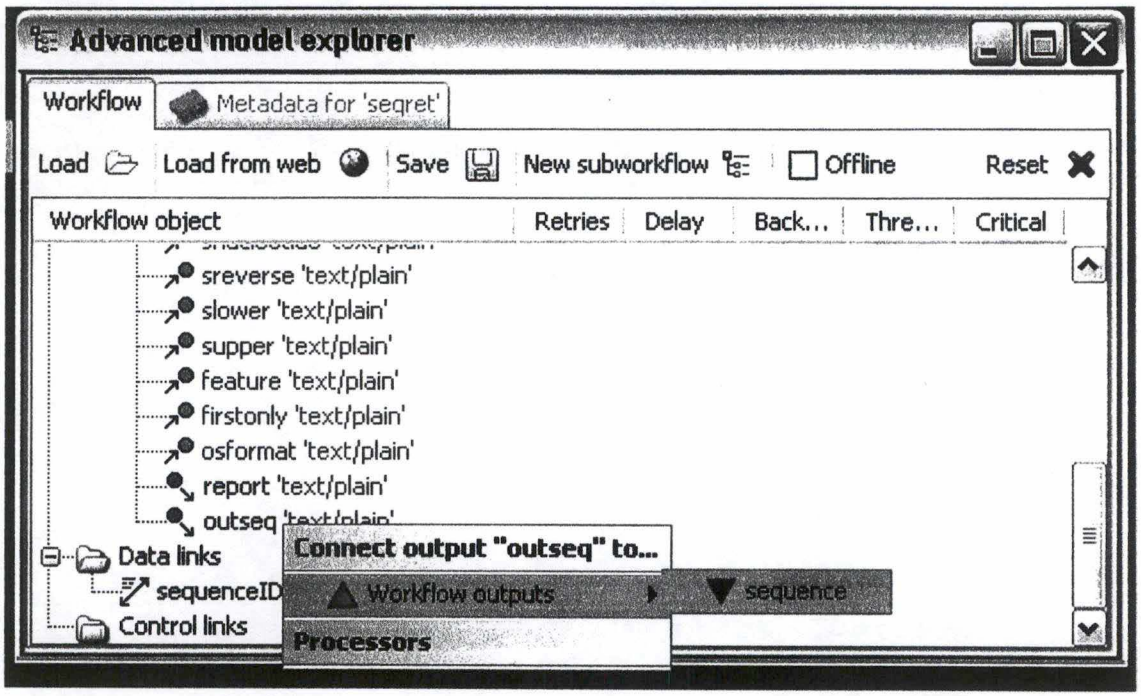

FIG. 3.12 – Création d'un workflow - « Liaison » des éléments

3.3 Point forts / faibles

Nous venons d'illustrer, à travers deux petits exemples, les bases du logiciel Taverna. Nous allons maintenant exposer quelques qualités et défauts de ce dernier. Pour commencer, Taverna se montre peu intuitif. Une lecture préalable du manuel utilisateur se révèle nécessaire pour apprendre à manipuler l'interface car certaines actions « réflexes » ne sont pas autorisées. Par exemple, ajouter un composant à un workflow ne peut pas se faire en effectuant un glisser / déposer depuis la liste des services vers la représentation graphique du workflow comme on pourrait être tenté de le faire. Cependant, une fois l'interface mieux maîtrisée, ce qui apparaissait comme des petits « défauts » se révèle vite être des choix intéressants effectués par les concepteurs du logiciel. Par exemple, au niveau de l'enchaînement des composants dans un workflow. Plutôt que de permettre de « lier » les composants de façon graphique dans la représentation du workflow, il faut passer par un menu contextuel disponible à partir de l'explorateur de modèles où sont listés tous les composants du workflow. En effet, dans Taverna, la représentation graphique du workflow n'est là qu'à titre illustratif et ne permet pas d'interactions avec le workflow. De plus, si lier graphiquement plusieurs services ne disposant chacun que d'un seul paramètre en entrée et un seul

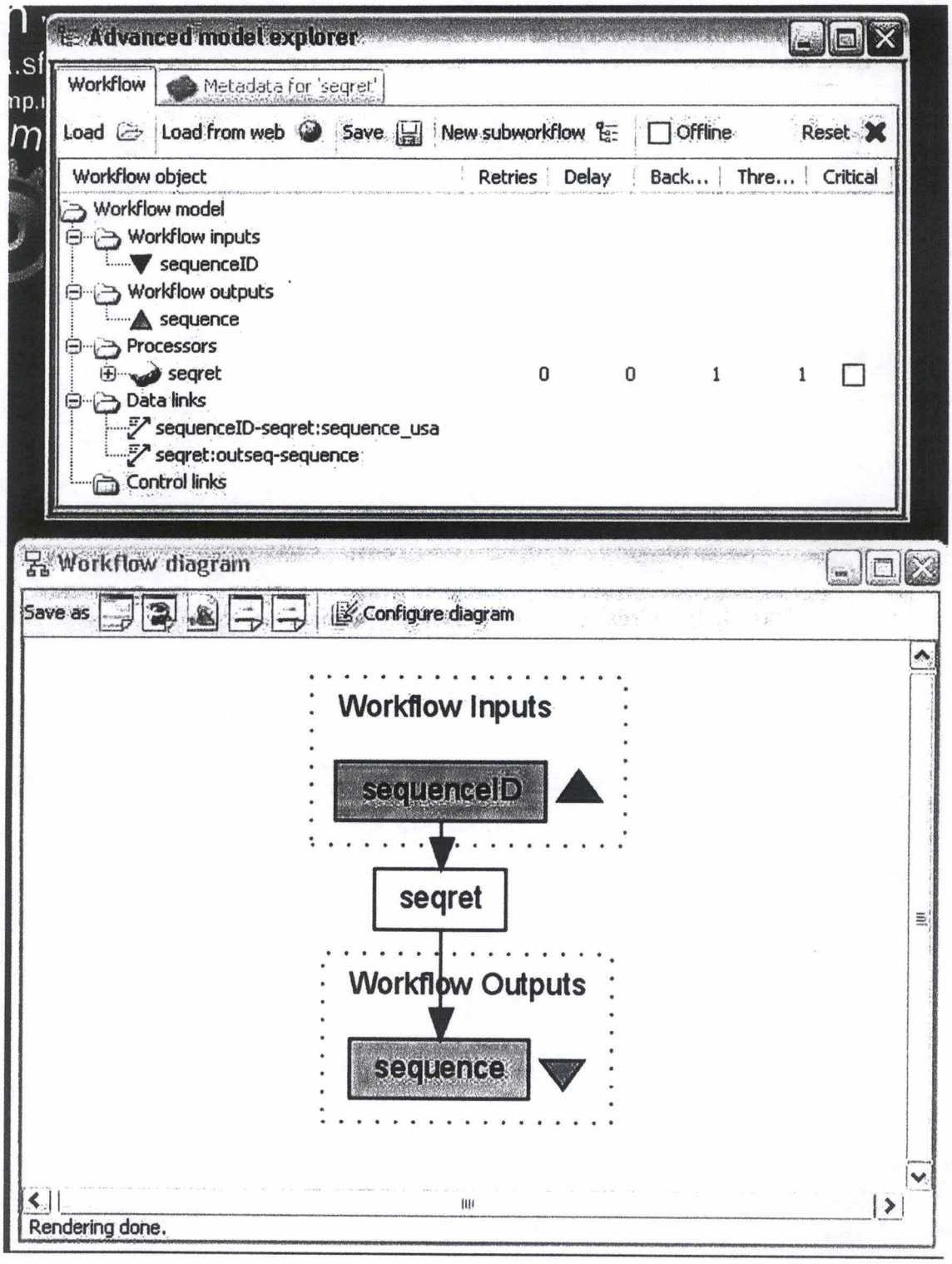

FIG. 3.13 – Création d'un workflow - Le workflow est prêt à être exécuté

paramètre en sortie aurait été assez simple à réaliser, force est de constater que de nombreux services disposent de plusieurs paramètres en entrée et fournissent plusieurs résultats. Lier graphiquement différents services de ce type aurait rendu obligatoire l'apparition d'un menu contextuel dans lequel il aurait été nécessaire de préciser le paramètre, ou le résultat, à lier, revenant donc au même principe que ce qui est proposé actuellement.

Un autre aspect « contre-intuitif » de l'interface survient lorsque l'on désire « exécuter » un workflow. En effet, plutôt que de présenter une boîte de dialogue dans laquelle entrer les valeurs des différents paramètres, il est obligatoire de « créer » des inputs (comme illustré à la figure 3.7) puis de leur attribuer des valeurs. Une fois l'habitude prise, on se rend compte que ce choix permet de conserver différentes valeurs pour les paramètres et de choisir parmi elles la valeur que l'on veut fournir au workflow, mais également de créer des listes de paramètres que l'on peut passer au workflow. Le mécanisme mis en place autorise également le chargement de valeurs depuis un fichier ou depuis une url. Il en résulte un moyen assez puissant de gérer les paramètres d'un workflow.

Un autre aspect concerne la liste des services disponibles. En effet, cette liste est présentée sous la forme d'une arborescence reprenant tous les services disponibles, comme illustré à la figure 3.2. Les services sont présentés par fournisseur et les services proposés ne disposent souvent même pas d'une petite description. Il en résulte que pour pourvoir utiliser un service, il faut en connaître le nom ou dumoins une partie. Ainsi, une recherche effectuée sur « alignement » ne renvoie aucun résultat alors qu'un service BLAST pourrait correspondre.

D'un point de vue plus technique, le logiciel Taverna voit son moteur de workflows construit autour d'une API extensible pour supporter différents types de services et peut donc assez facilement être adapté à de nouveaux types de services.

Des types MIME [10] peuvent également être attachés aux entrées et sorties d'un workflow de manière à fournir une indication sur le type de données attendues en entrée ainsi que sur la façon dont le navigateur de résultats (voir figure 3.9) de Taverna doit présenter les sorties. Cependant, Taverna ne dispose d'aucun mécanisme de vérification des types de données. C'est donc à l'utilisateur de veiller à faire attention que les données qu'il fournit sont bien du type attendu.

Un dernier inconvénient de Taverna est inhérent à son principe de fonctionnement. En effet, ce logiciel n'offre pas d'exécution « interactive » d'un workflow. Il est nécessaire de fournir tous les paramètres nécessaires dès le

[10] Multipurpose Internet Mail Extensions

début, puis d'exécuter le workflow et de récolter les résultats. Une fois le workflow démarré, il est cependant encore possible d'intervenir dans le workflow pour éventuellement modifier une donnée intermédiaire via un mécanisme de points d'arrêt. De part cette exécution non interactive du workflow, Taverna rend obligatoire la présentation des services au niveau de leurs opérations. Par exemple, un service constitué de trois étapes (initialisation, exécution, consultation) devra présenter ces trois opérations pour pouvoir être exécuté dans Taverna. En effet, imaginons que la phase d'initialisation nécessite de fournir un identifiant pour identifier les données, ensuite, la phase exécution reçoit de l'initialisation l'identifiant, mais attend les données à traiter, alors que la phase résultat utilise l'identifiant pour retourner les résultats obtenus. De façon interactive, le service aurait pu se constituer d'un seul composant, qui, au moment voulu, aurait demandé les paramètres nécessaires à son déroulement. Comme l'exécution d'un workflow dans Taverna se passe de façon non-interactive, il est obligatoire de séparer les différentes opérations du service afin de pouvoir fournir les données aux différentes étapes.

Taverna présente donc une excellente solution pour la création d'enchaînements complexes (ou non) de services bioinformatiques, ou autres, mais nécessite une période « d'apprentissage » et une adaptation de sa technique de travail avant d'être pleinement exploitable.

Chapitre 4

Babel

4.1 Introduction

Le Centre de Ressources INFOBIOGEN (CRI) a été créé le 10 juin 1999 par le Ministère Français de l'Education Nationale, de la Recherche et de la Technologie d'une part, et l'Université d'Evry Val d'Essonne d'autre part (Voir [Inf]).

Il constitue un centre national pour la recherche, le développement et l'exploitation de l'informatique appliquée à la Génomique. Le but de ce centre est d'exploiter et de développer des services bioinformatiques d'intérêt général afin d'en permettre l'utilisation par l'ensemble de la communauté scientifique française. Un accès aux différentes banques de données ainsi qu'à l'utilisation des environnements d'analyse et d'interrogation nécessaires pour l'exploitation de celles-ci est offert via Babel. Babel est un environnement Web qui permet la publication sur Internet d'applications bioinformatiques normalement exploitées en ligne de commande. Le serveur Babel est en fait basé sur un générateur automatique de fomulaires HTML construits à partir de programmes répondant à une syntaxe Unix type [1] et offre un accès aux programmes EMBOSS [2] ainsi qu'à un panel étendu de programmes classiques de bioinformatique (Blast, Fasta, etc.). La section suivante illustre le fonctionnement de Babel. Vient ensuite le relevé des avantages et inconvénients principaux de cet environnement Web.

[1] ligne de commande composée du nom du programme, d'une liste de paramètres et des entrées et sorties standards

[2] The European Molecular Biology OpenSoftware Suite

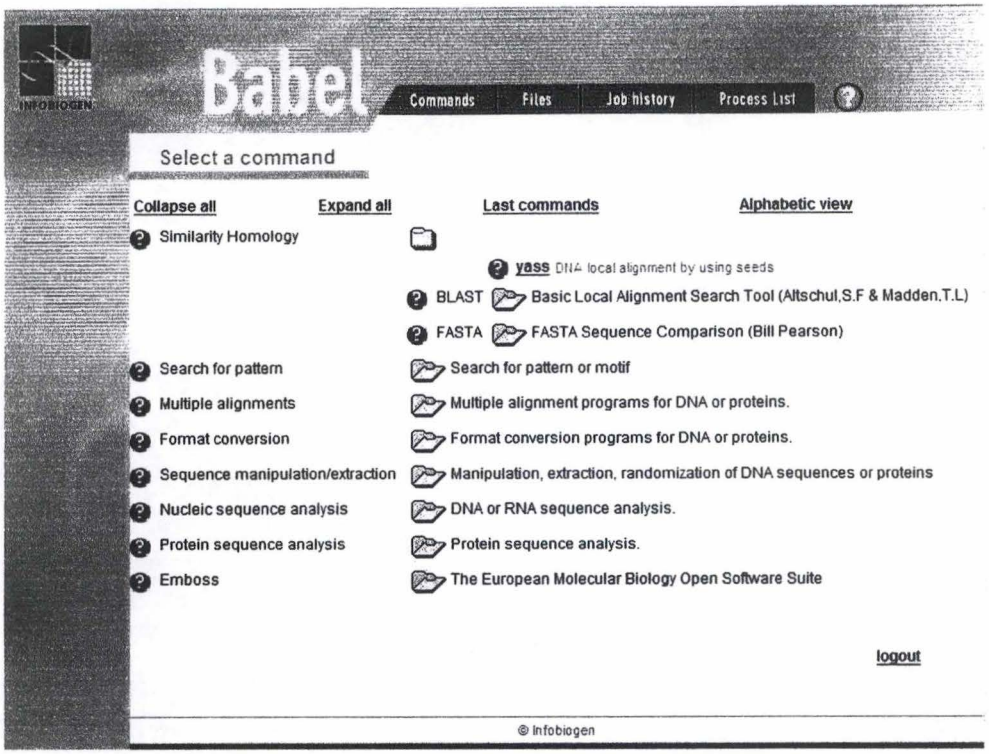

FIG. 4.1 – Arborescence des services disponibles

4.2 Fonctionnement

Pour utiliser Babel, deux possibilités sont offertes. On peut l'utiliser en mode anonyme ou en mode authentifié. En mode anonyme, la session de travail est automatiquement clôturée après un certain délai d'inactivité. Un espace temporaire de 200 Mb est alloué mais tout ce qu'il contient est effacé lorsque la session est clôturée. En mode authentifié, les sessions sont permanentes et l'espace de travail est porté à 300 Mb. Une fois le mode choisi, l'utilisateur se voit proposer une arborescence des services disponibles (voir figure 4.1. Cette arborescence est organisée par types de services. Chaque service est accompagné d'une courte description. Les services proposés sont également disponibles sous la forme d'une liste alphabétique.

Une fois le service choisi, un formulaire HTML est alors présenté à l'utilisateur avec les paramètres du services ainsi que les zones pour entrer les données. La figure 4.2 présente un exemple de formulaire.

Lorsque l'on demande l'exécution du service, on obtient alors un écran qui résume la requête et fournit, sous forme de fichier, les résultats de l'exécution. Ces fichiers sont stockés dans l'espace de travail dont dispose l'utilisateur.

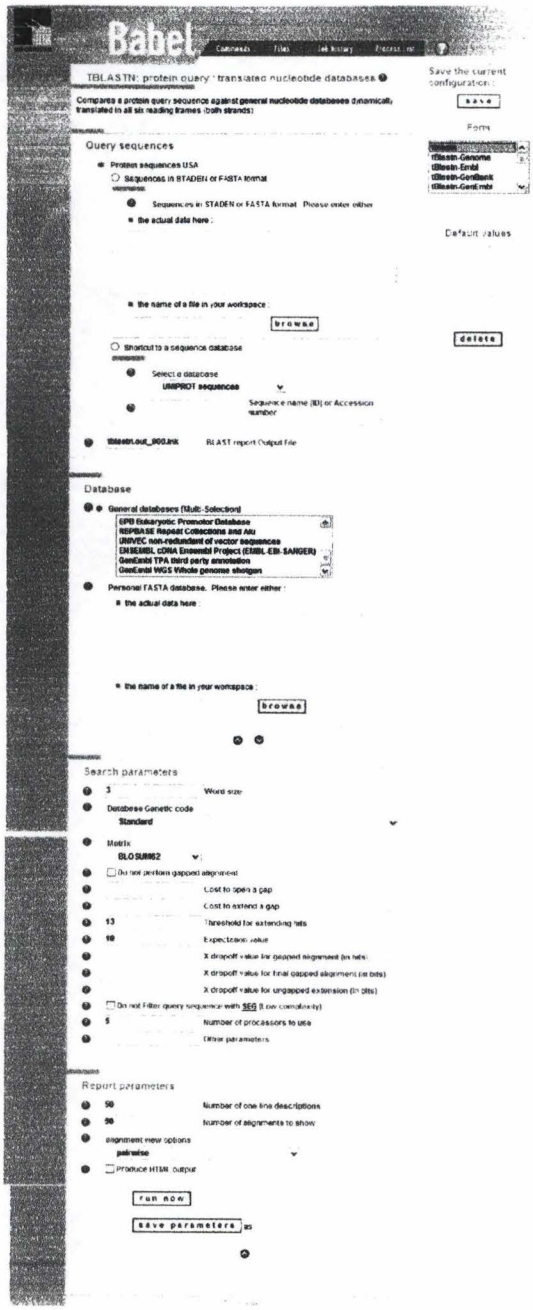

FIG. 4.2 – Exemple de formulaire pour un service

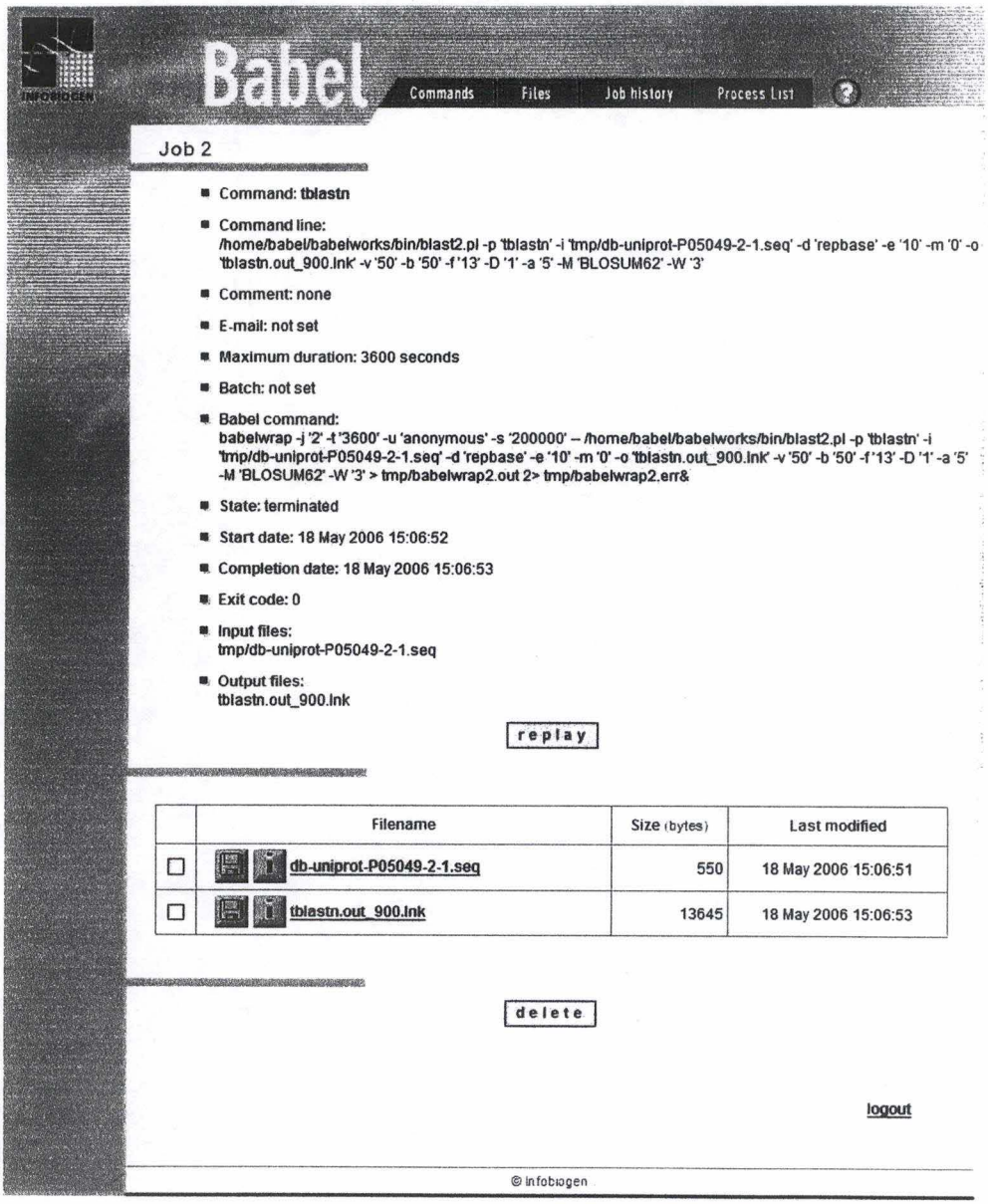

FIG. 4.3 – Résultat de l'exécution d'un service

4.3 Point forts / faibles

L'environnement Babel a été succintement présenté ci-avant. En voici quelques avantages et inconvénients.

Au niveau de son utilisation, Babel offre une grande facilité d'accès. Un simple navigateur suffit pour se connecter et utiliser l'environnement. Babel offre également, pour tous les services bioinformatiques proposés, des interfaces homogènes et conviviales, les formulaires étant générés de façon automatique, ce qui assure l'homogénéité, mais sont traités manuellement par après pour garantir la convivialité.

Les services sont présentés de la façon la plus complète qui soit, puisque chaque paramètre du service est repris dans le formulaire. Un aspect positif est que les formulaires sont générés de façon automatique mais sont ensuite complétés manuellement comme précisé ci-dessus. Chaque paramètre d'un service se voit alors accompagné d'une courte description et une page d'aide expliquant le fonctionnement du service est également disponible.

Le fait de présenter tous les paramètres d'un service sur le même formulaire rend cependant ce dernier moins lisible et plus complexe, même si chaque paramètre dispose d'une petite explication.

Un autre atout de Babel est d'offrir un espace personnalisé où il est possible de stocker ses données. Cet espace permet dès lors d'utiliser les données issues d'un service comme entrées pour un autre service sans avoir besoin de procéder à plusieurs téléchargements. L'espace personnel peut également servir pour sauvegarder des résultats d'analyses ou enregistrer des paramètres pour pouvoir les réutiliser ultérieurement. De plus, comme seul un navigateur est nécessaire, ces données sont accessibles de presque partout.

Babel autorise également l'exécution de services qui demandent de longs temps d'exécution. A cette fin, un mécanisme de gestion de « jobs » est mis en place et permet de consulter à tout moment l'état des services demandés.

Babel se présente donc comme une solution simple pour exécuter des services bioinformatiques avec toute l'aide nécessaire à portée de clic. L'environnement est assez facilement maîtrisable grâce à l'homogénéité des interfaces fournies et à l'ergonomie générale du site. Petite ombre au tableau, il n'est pas possible d'effectuer une recherche de services. Il faut donc soit connaître le service que l'on veut utiliser et espérer qu'il soit disponnible, soit chercher dans la liste des services, en se servant des descriptions, un service correspondant à ce que l'on veut effectuer.

Chapitre 5

BIGRE

5.1 Introduction

Le projet BIGRE[1] est né de « *la volonté de simplifier l'utilisation des outils bioinformatiques aux différents utilisateurs potentiels, de faciliter le développement de futurs services et d'assurer une utilisation plus efficace des ressources disponibles* [2] ».

La plateform BIGRE est architecturée autour d'un système distribué et est composée de trois types d'éléments : les clients, les médiateurs et les services (voir figure 5.1) :

Le client constitue l'application utilisateur principale. Il a pour charge de fournir à l'utilisateur une interface homogène, cohérente et adaptée à son profil, d'accès aux différents *services* par l'intermédiaire des médiateurs.

Les services sont représentés soit par des composants façades (ou *wrappers*) pour des outils existants, soit par des services natifs, c'est-à-dire directement développés pour être intégrés à BIGRE.

Les médiateurs ont pour charge d'assurer les communications entre *clients* et *services*. Ils sont organisés en une fédération et forment un système distribué. Ils permettent, entre autre, la découverte dynamique de services, la sécurité des communications et des accès, la facturation des services[3], etc.

---

[1] BIoinformatics Grid Ressources and Environments (voir `http://www.info.fundp.ac.be/~bigre/`

[2] Voir [BD03]

[3] Dans le cadre de services payants

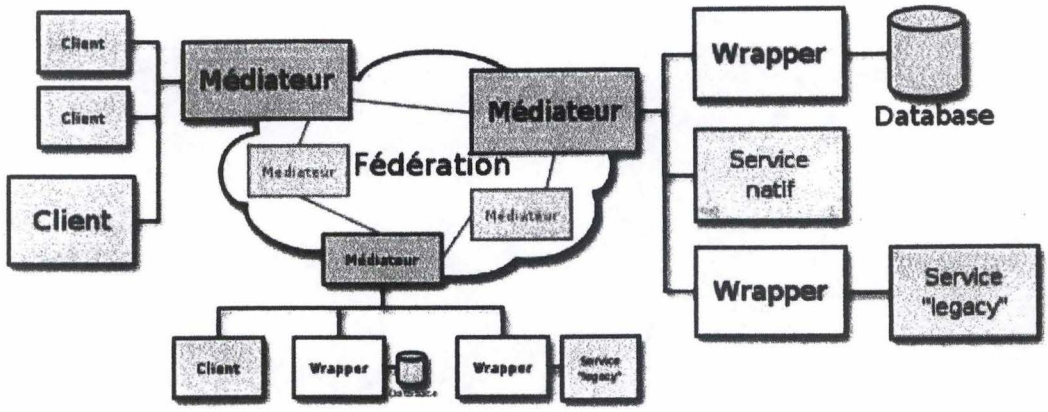

FIG. 5.1 – Architecture de la plateforme BIGRE [DBDM04]

5.2 Fonctionnement

De façon simplifiée, l'utilisation de la plateforme BIGRE se déroule de la façon suivante :

1. L'utilisateur se connecte, via le client, à son médiateur
2. Une fois connecté, il reçoit une liste de services ou peut effectuer une recherche. Dans ce cas, son médiateur lui retourne les références des services correspondant à sa recherche.
3. L'utilisateur choisit alors le service qu'il veut utiliser et effectue les opérations nécessaires à la réalisation du traitement proposé par le service.

L'explication du déroulement est ici fortement simplifiée. N'y sont pas décrites les informations de sécurisation, de qualité de services, etc. Le but n'est ici que de schématiser l'utilisation d'un service via la plateforme BIGRE, du point de vue du client.

La plateforme BIGRE agit comme un intermédiaire entre le client et les services. En effet, BIGRE n'agit pas au niveau de l'exécution du service mais s'occupe de relayer les informations entre le client et le service tout en assurant la gestion et la sécurisation de cette connexion.

Un premier client pour BIGRE a été développé. Architecturé sur la plateforme Eclipse RCP[4] il offre une interface utilisateur homogène pour l'exécution des différents services disponibles dans BIGRE. Tout en profitant des facilités fournies par la plateforme Eclipse (Copier / Coller, Edition, Profils multiples, etc.) le client offre l'avantage de pouvoir s'adapter automatiquement à un type de données ou de services via l'utilisation de plugins de

[4]Eclipse Rich Client Platform (Voir `http://www.eclipse.org`)

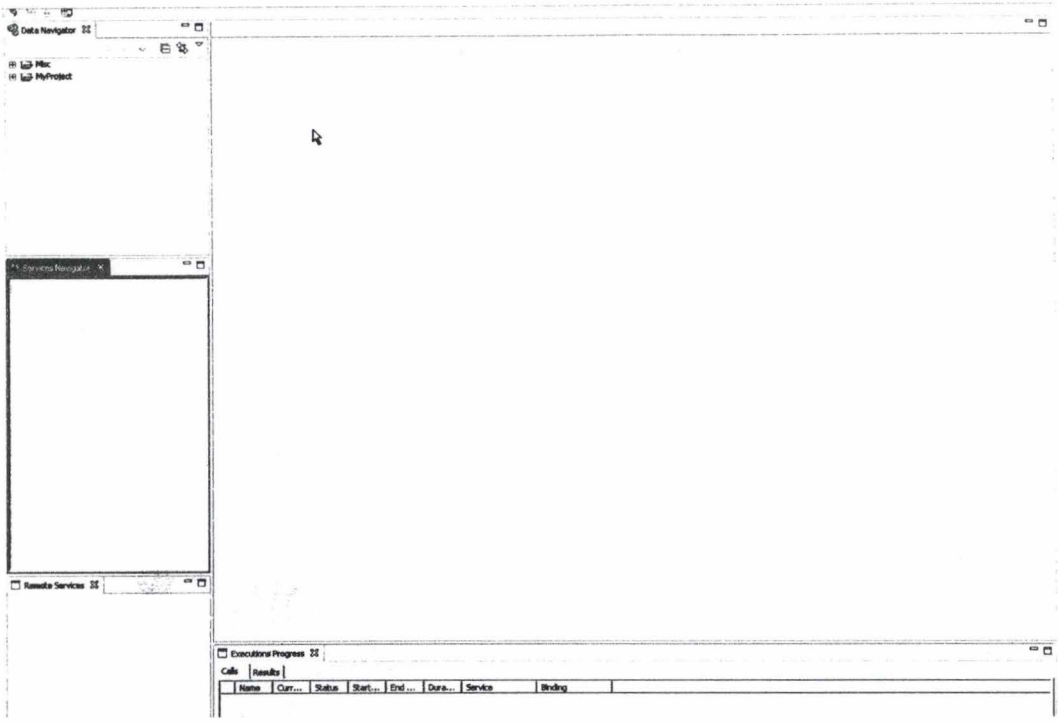

FIG. 5.2 – Client BIGRE

visualisation et d'édition chargés dynamiquement. Un editeur / visualiseur générique est proposé par défaut pour s'adapter rapidement à n'importe quel nouveau type de données.

L'interface proposée par ce client est illustrée à la figure 5.2.

L'utilisateur commence par choisir son environnement de travail en fonction de son profil, comme illustré à la figure 5.3.

La liste des services disponibles est alors téléchargée et proposée à l'utilisateur. Une fois le service choisi, il est rendu disponible dans un navigateur de services regroupant les différents services par leur type (Voir figure 5.4).

L'utilisateur peut alors exécuter le service qu'il a choisi. Une fenêtre apparaît alors lui demandant de fournir les paramètres nécessaires à l'utilisation du service comme le montre la figure 5.5.

Le service peut alors être exécuté. Plusieurs services peuvent être exécutés et leur déroulement peut être suivi au moyen d'une fenêtre reprenant l'état des différents jobs comme illustré à la figure 5.6.

Lorsque l'utilisateur décide de consulter le résultat d'un service dont il a demandé l'exécution, différents choix lui sont proposés concernant la façon dont il veut que les données soit représentées (Voir figure 5.7).

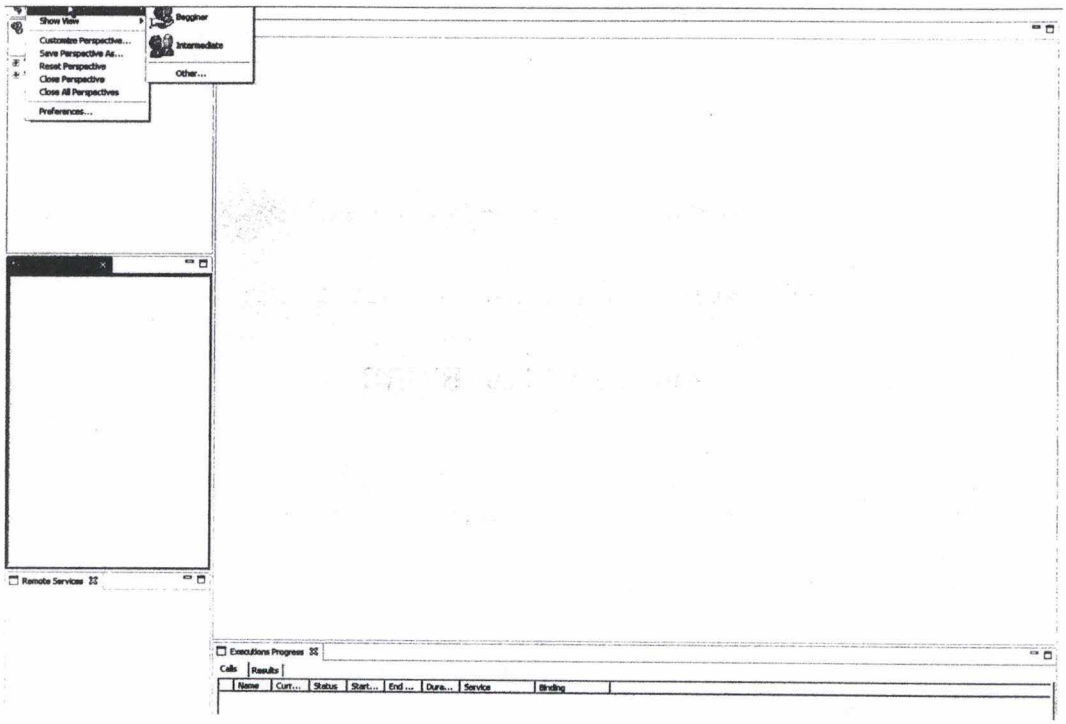

FIG. 5.3 – Client BIGRE : Choix du profil

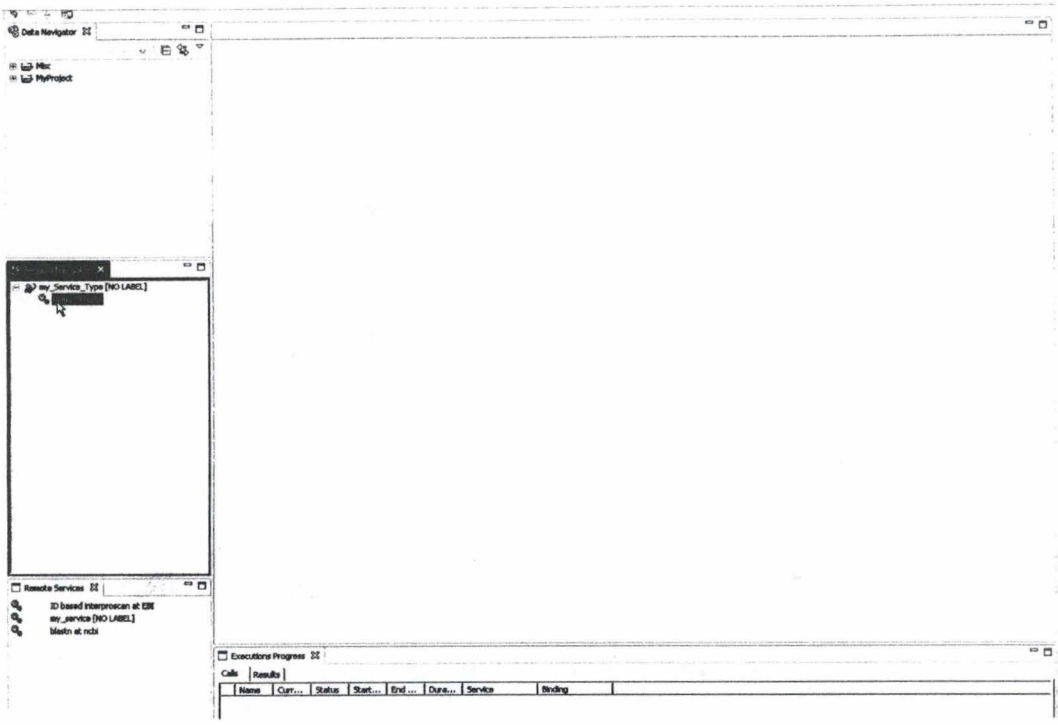

FIG. 5.4 – Client BIGRE : Choix du service

Le résultat de l'exécution du service est alors présenté en fonction du choix de présentation effectué comme cela est illustré à la figure 5.8.

Le client offre également la possibilité de créer un scénario d'exécution pour un service comme le montre la figure 5.9.

5.3 Point forts / faibles

L'avantage premier du client présenté à la section précédente est de fournir une interface homogène pour l'exécution des différents services. Un choix est également offert à l'utilisateur lui permettant de choisir la « perspective[5] » de l'interface en fonction de son « niveau » tel que novice, expert, etc. L'interface présente les différents services disponible dans BIGRE dans une arborescence et permet également l'exécution de différents services de façon simultanée. Un aspect très intéressant de l'interface est de proposer différentes « vues » permettant de contrôler et de gérer ses données, ses services utilisés, les jobs en cours, les résultats obtenus, etc. Cette présentation se révèle cependant peu intuitive et nécessite un temps d'adaptation, surtout lorsque l'utilisateur

[5]Organisation des différents éléments graphiques qui composent l'interface

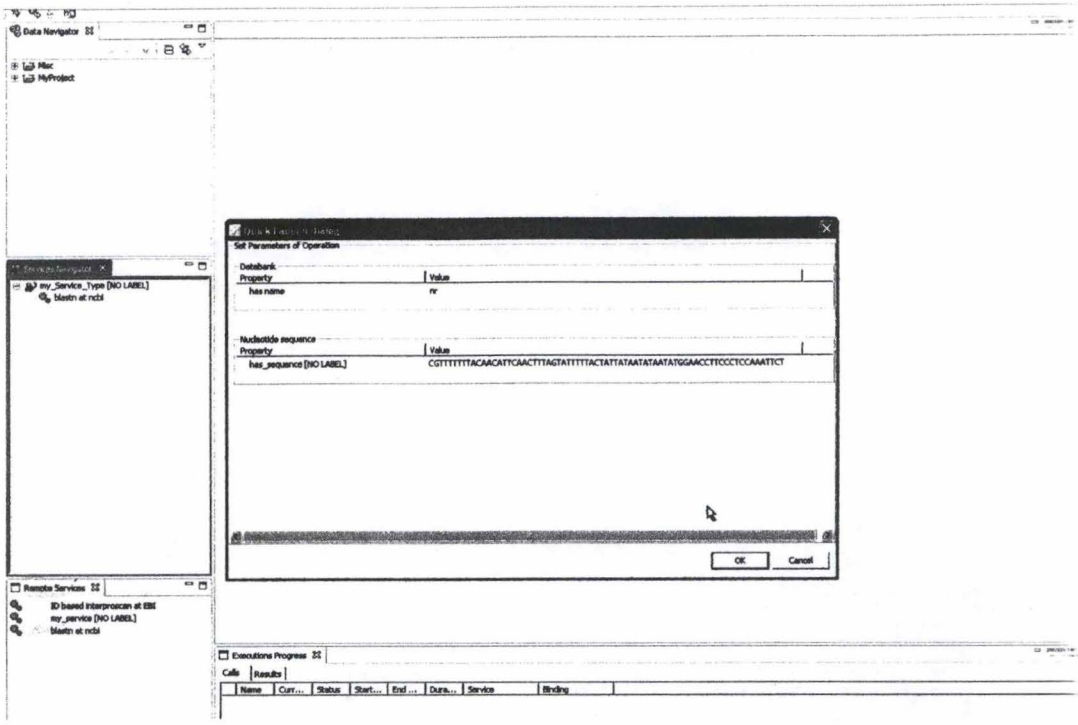

FIG. 5.5 – Client BIGRE : Paramètres du service

est novice. En effet, voir le service que l'on utilise réduit à une ligne dans une liste de jobs en cours peu se révéler un peu déroutant. La méthode pour saisir les paramètres lors de l'exécution d'un service se révèle également peu conviviale ainsi que le fait de devoir choisir le mécanisme d'affichage des résultats, bien que cette option présente l'avantage d'offrir à l'utilisateur le choix de la façon d'afficher les résultats obtenus.

Un autre aspect positif réside dans la possibilité de créer des scénarios permettant d'automatiser l'exécution de divers processus.

Différents avantages de la plateforme BIGRE sont cependant indépendants du client. On retiendra notamment, la sécurisation des transferts de données. En effet, la plupart des services bioinformatiques accessibles par Internet ne sont pas sécurisés, il en résulte que les données circulent en clair et peuvent donc être facilement interceptées. Cela constitue un sérieux handicap dans le cadre de recherches privées pour lesquelles les chercheurs désirent garder la confidentialité des résultats et des données traitées. BIGRE apporte une solution à ce problème en offrant de sécuriser le chemin de communication entre le client et le service, garantissant ainsi la sécurité des données.

La plateforme BIGRE offre également, via sa fédération de médiateurs, un moyen d'effectuer une recherche dynamique de services. Ce système permet

FIG. 5.6 – Client BIGRE : Jobs en cours

qu'un nouveau service soit rendu très rapidement accessible aux clients de la plateforme mais également qu'un utilisateur puisse accéder à un service qu'il ne connaissait pas jusqu'alors.

A un niveau plus technique, la plateforme BIGRE est architecturée autour d'un système distribué. Cela lui permet ainsi d'évoluer assez facilement, que ce soit d'une façon quantitative (montée en charge) ou qualitative (changement de langage, modification de l'architecture, ...). Le fait d'être construit sur un tel système permet également à BIGRE de supporter une grande hétérogénéité de ses composants, en utilisant conjointement des composants pré-existants, des composants natifs (développés spécialement pour BIGRE) ou encore des composants de tiers, éventuellement payants, tout en offrant à l'utilisateur l'impression qu'il s'agit d'un système unique et intégré. L'achitecture distribuée permet également un partage des ressources et un contrôle de l'accès à ces dernières.

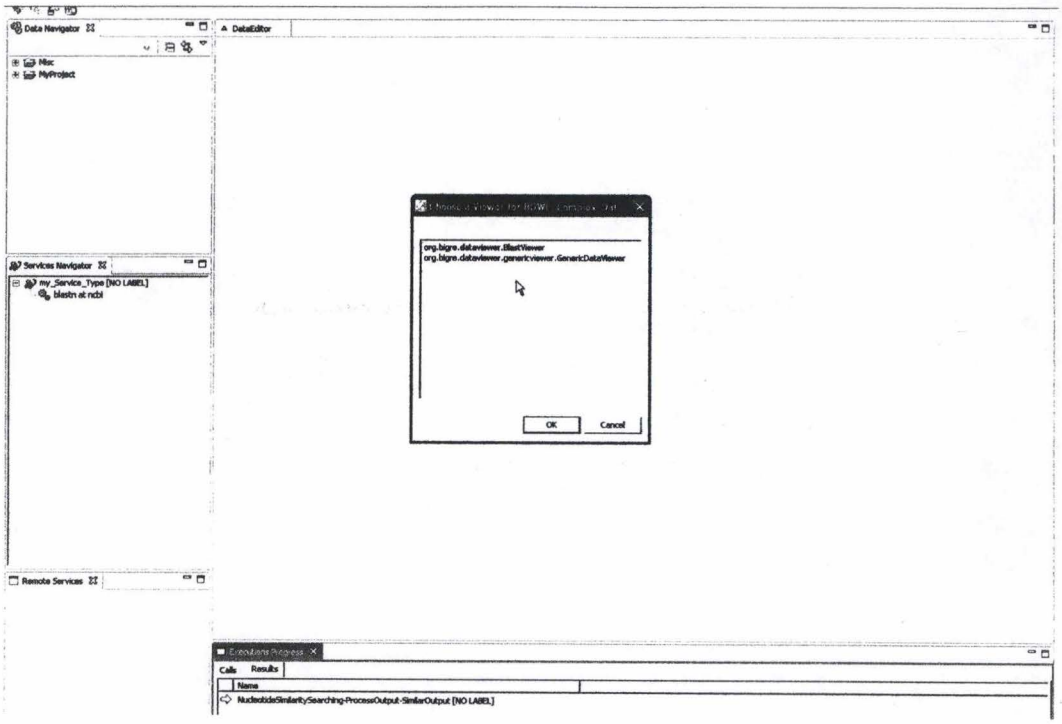

FIG. 5.7 – Client BIGRE : Choix de la représentation

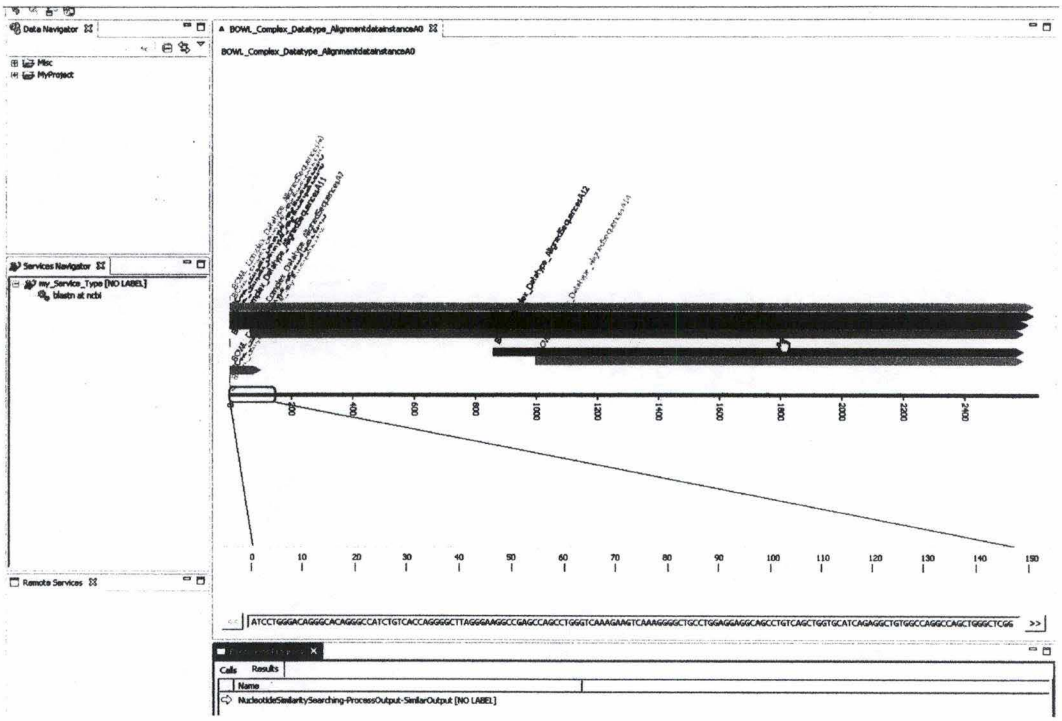

FIG. 5.8 – Client BIGRE : Affichage du résultat

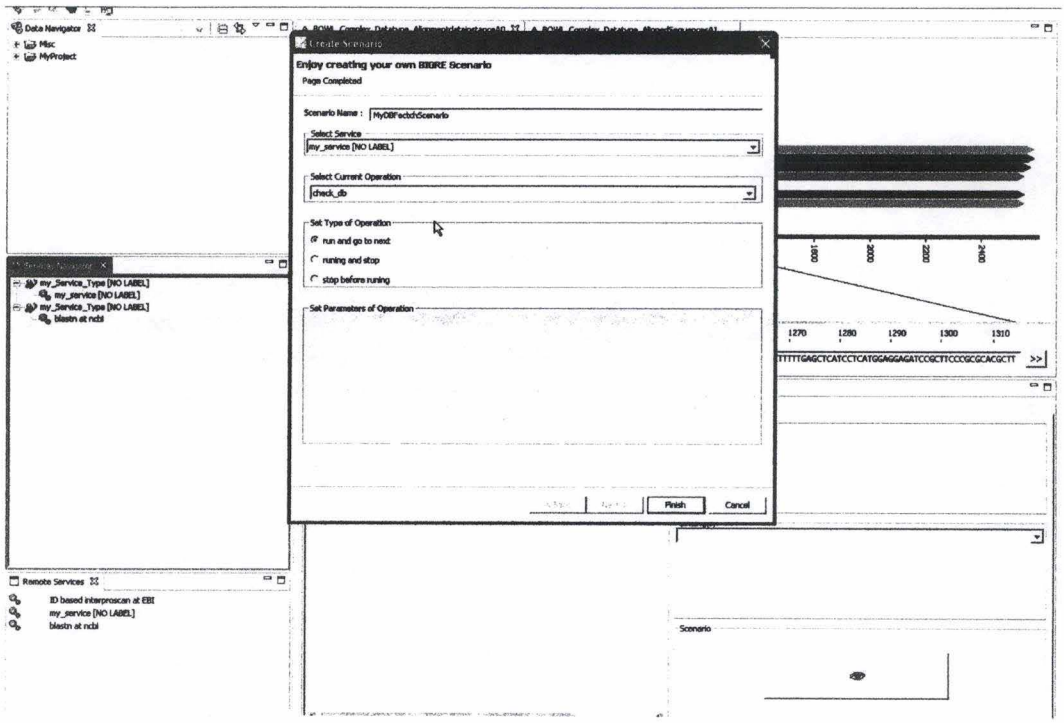

FIG. 5.9 – Client BIGRE : Réalisation d'un scénario

Chapitre 6

Synthèse

Nous venons de voir, à travers trois exemples, plusieurs méthodes pour palier aux problèmes de disparité et d'hétérogénéité des services bioinformatiques. En effet, chacune des solutions présentées précédemment répond à sa manière aux problèmes soulevés. Un autre projet, appelé Bioside [1], avait également retenu notre attention. Malheureusement peu d'informations sont disponibles à son sujet et nous n'avons donc pu le décrire ici.

Dans ce qui a été décrit, trois types de solutions sont envisagées. La solution mise en place par le logiciel Taverna regroupe une multitude de services bioinformatiques et permet de les enchaîner les uns aux autres pour former des workflows, tout en donnant l'impression que tous les services sont accessibles localement alors qu'ils restent dispersés à travers le monde. Comme Taverna se concentre sur la construction de workflows, l'hétérogénéité des services n'est également plus un obstacle étant donné que les services sont encapsulés dans les composants d'un workflow et ne sont plus utilisés en tant que tel.

Une autre méthode est employée par l'environnement Babel qui regroupe différents services bioinformatiques sur un serveur et qui offre un accès à ces différents services via des formulaires HTML générés de façon automatique. Les différents services sont accessibles en un seul point et l'hétérogénéité des services disparait au profit d'interfaces homogènes créés par le générateur automatique.

Le système BIGRE, quant à lui, se positionne à l'intersection des deux solutions précédentes en offrant un accès centralisé à différents services (seul l'accès est centralisé, car les services restent distants) tout en ayant pour but de fournir des interfaces homogènes pour les différents services proposés.

Un autre aspect qui sépare ces différentes solutions, outre leur mode de

[1] `http://departements.enst-bretagne.fr/lussi/article.php3?id_article=55`

fonctionnement, concerne leur mode d'utilisation. BIGRE et Taverna sont articulés autour d'un client « lourd », en ce sens qu'un programme doit être déployé sur une machine pour pouvoir utiliser ces systèmes. Babel, à l'opposé, est architecturé autour d'un client « léger », c'est à dire, qui ne nécessite pas de déployer un programme supplémentaire étant donné que Babel est accessible via un simple navigateur.

L'avantage d'une solution mettant en place un client « léger » tel que Babel est de rendre le système accessible et de permettre aux utilisateurs d'accéder à leurs données depuis presque partout, tandis que des clients « lourds » tels que ceux proposés par Taverna et, plus tard, par BIGRE nécessitent de se connecter depuis une machine disposant du client et même, si les utilisateurs veulent utiliser leurs données, depuis la machine qui les contient.

D'un autre côté, la mise en place d'un système reposant sur un client « léger » nécessite que le système dispose de ressources suffisantes pour gérer tous les utilisateurs et rend le système moins approprié pour résister à une forte montée en charge. Qui plus est, si plusieurs points d'accès ne sont pas mis en place, le système risque de rapidement se révéler inapte à accueillir de nombreux clients simultanément.

Les systèmes architecturés autour de clients « lourds » sont eux bien plus aptes à résister à une forte montée en charge et sont capables d'accueillir simultanément de nombreux clients étant donné que chaque client utilise ses propres ressources. Grâce à ce dernier point, mais également parce qu'il ne sont pas tenus par l'utilisation d'un navigateur, les clients « lourds » peuvent également être beaucoup plus riches en terme de fonctionnalités.

Deuxième partie

Analyse

Chapitre 7

Méthode

Le but de ce mémoire consiste en la création d'un client pour BIGRE. L'objectif est d'obtenir un client capable de permettre l'exécution des services bioinformatiques fournis par l'intermédiaire de BIGRE.

Les contraintes posées sont que le client doit être capable de générer des interfaces graphiques de façon dynamique sur base de la description du service, fournie par BIGRE, que ces interfaces doivent gérer le flux d'exécution du service (ordonnancement des différentes étapes d'un service), mais également qu'elles doivent être adaptables aux préférences de l'utilisateur (style habituel, agencement modifiable, etc.). D'un point de vue non-fonctionnel, le client doit être évolutif et facilement modifiable afin d'être facilement adaptable à d'éventuelles modification de l'architecture de BIGRE.

La méthode employée pour la réalisation du client se rapproche d'une méthode par prototypage. Sur base des exigences décrites précédemment, une première approche a été réalisée en essayant concrétiser ces besoins et d'y apporter des idées de solution. Sur cette base, une première architecture a été développée. Un prototype reprenant un sous-ensemble critique de cette architecture a alors été implémenté. Sur base du prototype développé, l'architecture a alors été retravaillée pour corriger ou améliorer certains aspects du système et le prototype a ensuite été réadapté afin de valider la nouvelle architecture. Cette méthode semblait la plus adaptée car les spécifications étant très « larges », il était nécessaire de rapidement développer quelque chose de concret pour valider la faisabilité des idées développées.

L'architecture qui est présentée au chapitre 9 n'a donc pas été pensée telle-quelle mais constitue la synthèse de plusieurs « essais-erreurs ».

Dans les chapitres suivants sont présentés l'architecture réalisée ainsi que les détails importants de l'implémentation du prototype. Un exemple illustre alors le fonctionnement de ce prototype.

Chapitre 8

Analyse

8.1 Introduction

Ce chapitre a pour objectif de présenter le raisonnement qui a servi de base à la réalisation de l'architecture. Pour commencer différents « besoins » sont décrits et constitue une concrétisation des exigences succintement décrites dans le chapitre précédent. Des ébauches de solutions sont ensuite présentées.

8.2 Besoins

L'objectif de cette analyse consiste en la réalisation d'un client pour le système BIGRE devant permettre d'exécuter les services fournis par l'intermédiaire de ce dernier. Comme expliqué au chapitre 5, BIGRE se « contente » de fournir un moyen de communication avec le service, et d'assurer ces communications. Concrètement, BIGRE fournit au client une description du service respectant un format précis décrit ci-dessous, puis répercute les actions effectuées par le client sur le service et retourne alors les résultats fournis par le service au client.

La description d'un service est fournie dans un format texte qui respecte l'ontologie de services de BIGRE, illustrée à la figure 8.1 et disponible à l'adresse suivante : `http://www.info.fundp.ac.be/~bigre/ontologies/services.owl`.

En informatique, une ontologie est « *un ensemble structuré de concepts. Les concepts sont organisés dans un graphe dont les relations peuvent être des relations sémantiques ou des relations de composition et d'héritage (au sens objet). L'objectif premier d'une ontologie est de modéliser un ensemble de connaissances dans un domaine donné* [Wik]. Une ontologie permet donc de décrire les objets d'un domaine et de décrire les relations qui existent entre

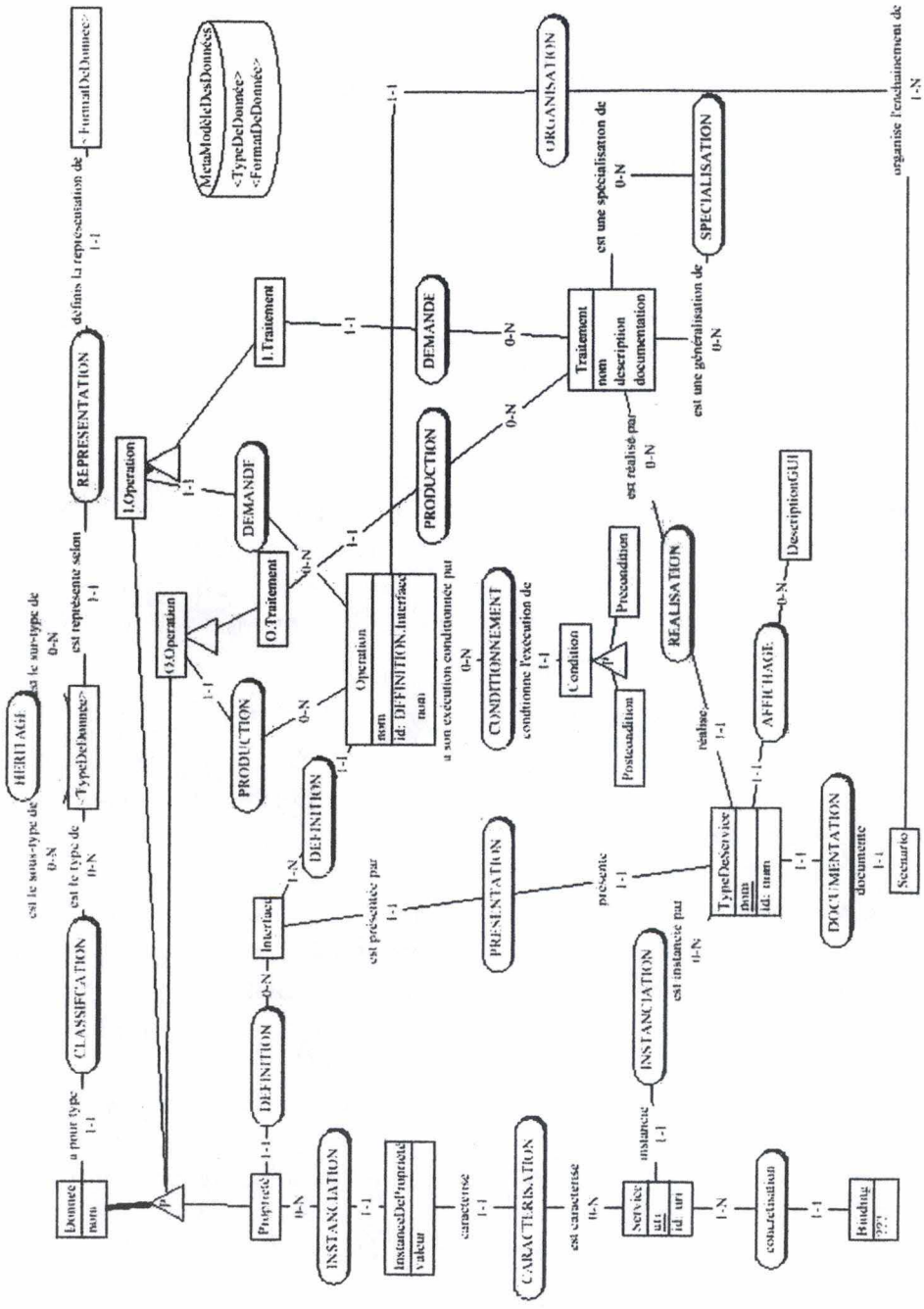

FIG. 8.1 – Ontologie de Bigre [DBDM04]

ces objets .

Dans le cadre de BIGRE, la description d'un service est créé selon l'ontologie définie à la figure 8.1. Sur base de cette ontologie, il est dès lors possible d'analyser la description d'un service pour savoir quels sont les objets qui constituent le service et quelles sont les relations entre ces objets. Autrement dit, cela permet de savoir comment le service se comporte, c'est à dire, savoir ce qu'il réalise, quelles sont les opérations qu'il propose et quels sont leurs entrées et sorties, etc.

Dans cette ontologie, le concept de *Service* représente un service, une application, partagée dans BIGRE. Un tel *Service* réalise un *Traitement.* Un *Traitement* est un processus qui reçoit un ensemble de données en entrée, les transforme, et fournit les données transformées en sortie. Pour réaliser le *Traitement* proposé par un *Service*, il faut exécuter une ou plusieurs *Opérations.* Une *Opération* représente une action qu'un utilisateur peut effectuer auprès du service et s'exécute comme un tout, sans qu'aucun contrôle de son déroulement ne soit effectué. Elle est caractérisée par un nom, par un ensemble de données reçues en entrée, un ensemble de données fournies en sortie, un ensemble de pré-conditions qui doivent être remplies pour permettre l'exécution de l'opération ainsi qu'un ensemble de post-conditions qui sont vérifiées après exécution de l'opération. Les différentes *Opérations* fournies par un *Service* pour exécuter un *Traitement* sont organisées par un *Scénario* qui en définit l'ordre et l'enchaînement.

Sur base de ces concepts, le client doit afficher une interface permettant d'exécuter le service. La première étape consiste donc à analyser la description du service et à en extraire toutes les informations nécessaires.

Une fois ces informations extraites, le client a alors en charge la construction de l'interface graphique avec laquelle l'utilisateur pourra interagir. Un des objectifs du projet BIGRE étant de fournir des interfaces homogènes mais adaptables aux préférences de l'utilisateur, il est ici nécessaire de mettre en place un mécanisme capable de construire les interfaces selon ces préférences. Il faut donc que ce mécanimse soit capable de fabriquer des interfaces de différents types (formulaire, assistant, onglets, etc.) mais également qu'il soit capable de construire d'éventuels nouveaux types d'interfaces sans nécessiter de nombreuses modifications et ce pour respecter la contrainte d'évolutivité.

Une fois le mécanisme de création d'interfaces mis en place, il est également nécessaire de mettre en place un mécanisme pour que cette interface soit « réactive » mais également qu'elle respecte l'ordonnancement du service (ordre dans lequel peuvent être exécutées les opérations). Il faut donc mettre en place un mécanisme capable d'assurer l'ordre d'exécution des opérations.

Lorsque cet ordre d'exécution est assuré, il ne reste plus au client qu'à assurer la communication avec le système BIGRE, plus précisément un mé-

diateur, afin que l'interface corresponde bien à l'exécution du service, en ce sens que lorsque l'utilisateur demande l'exécution d'une opération du service, celle-ci soit exécutée.

Ce dernier point soulève également un autre problème qui est celui de la durée d'exécution d'une opération. En effet, il se pourrait qu'un service demande un long temps d'exécution pouvant, par exemple, s'étaler sur plusieurs jours. Il est alors impensable de « bloquer » le client durant toute la durée de cette opération. Il est dès lors nécessaire de mettre en place un mécanisme permettant l'exécution asynchrone d'un service, c'est à dire, un mécanisme permettant d'exécuter une opération, de quitter le client, puis de se reconnecter plus tard et de récupérer les résultats de cette opération en reprenant le fil d'exécution du service là où il avait été interrompu.

8.3 Solutions

Différents besoins ont été exprimés, de façon très générale, dans la section précédente. Voici maintenant les idées de solutions qui ont été mises en place pour les satisfaire.

8.3.1 Construction des interfaces graphiques

La première étape consiste à analyser la description d'un service. Un composant se charge de cette analyse et fournit les informations nécessaires pour la création de l'interface et l'utilisation du service. Cependant, il est nécessaire de trouver un moyen pour organiser les informations extraites de la description d'un service afin que les interfaces générées soient fidèles au service qu'elles vont représenter.

L'ontologie de BIGRE nous apprend qu'un service réalise un traitement et que, pour réaliser ce traitement, le service peut être découpé en plusieurs opérations atomiques, c'est à dire non découpable en d'autres opérations. Chaque opération possède un certain nombre d'inputs et fournit un certain nombre d'outputs. Un scénario décrit l'ordonnancement de ces opérations.

Le plus bas niveau où peut intervenir l'utilisateur dans l'exécution d'un service est donc le niveau des opérations. Il faut donc que l'interface générée présente les différentes opérations du services et donc la structuration des informations extraites de la description d'un service peut se faire au niveau des opérations qui représenteront donc des « unités » graphiques de l'interface. C'est à dire les plus petits ensemble d'éléments indissociables de l'interface graphique du service.

Le premier problème qui se pose consiste en la création d'un mécanisme de construction des interfaces qui soit adaptable aux préférences des utilisateurs mais également facilement modifiable pour pouvoir accepter de nouveaux types d'interfaces.

Pour créer ce mécanisme, la construction d'une interface graphique a été découpée en deux étapes. La première étape consiste en la création des éléments graphiques de base de l'interface, appelés widgets. Ces éléments sont, par exemple, un bouton, un champ d'édition, une case à cocher, etc. Une fois ces éléments créés, il reste à les assembler et à leur donner l'aspect voulu pour l'interface. Les différents widgets peuvent, par exemple, être assemblés à la façon d'un formulaire, les uns au dessus des autres, ou encore sous forme d'assistant, comme une suite de plusieurs écrans présentant les différentes sections de l'interface.

Pour chacune de ces deux étapes, les contraintes sont identiques aux contraintes globales de l'interface, à savoir l'adaptabilité et la modulabilité.

La solution pour résoudre ce problème est issue du modèle de conception[1] dit de « la fabrique abstraite ». Le principe général du modèle de la fabrique abstraite est que le système fait appel à un composant spécial, appelé fabrique abstraite car elle ne construit pas elle même d'objets, pour demander la construction d'un objet. La fabrique utilise alors la fabrique concrète appropriée pour construire l'objet demandé.

Ce modèle de conception permet de répondre efficacement au besoin d'adaptabilité puisque, en procédant de la sorte, les widgets et interfaces pourront être construits selon les préférences de l'utilisateur. En effet, en fonction des choix de l'utilisateur, le système demandera la construction des widgets à la fabrique correpondant aux choix effectués, idem pour les interfaces. Cette solution présente l'avantage d'également répondre au besoin de modularité puisqu'il suffit d'ajouter une fabrique concrète supplémentaire pour ajouter un nouveau type d'objets.

8.3.2 Scénario du service

Une fois le mécanisme de création des composants graphiques et d'assemblage de ces composants mis en place, il reste à mettre en place un mécanisme pour gérer l'ordonnancement du service, à savoir l'ordre dans lequel vont pouvoir s'exécuter les différentes opérations proposées par un service. Comme cela a été vu au point 8.3.1, l'analyse de la description d'un service permet de structurer les informations d'un service d'après ses opérations. Il suffit dès lors de mettre en place un composant qui va s'occuper de gérer

[1] Design Patern [GHJV99]

ce scénario et de ne permettre à l'utilisateur d'exécuter que les opérations exécutables en fonction du scénario. Comme le scénario doit correspondre à l'exécution réelle du service, ce composant aura également la charge d'appeler un composant chargé de la communication avec le système BIGRE et de s'assurer qu'une opération aura bien été effectuée avant de permettre la suivante dans le scénario.

8.3.3 Gestion des services

Comme cela a été vu à la section 8.2, il est nécessaire d'envisager le cas où l'exécution d'un service pourrait s'étaler sur un long laps de temps. A priori, les opérations sont « bloquantes ». Autrement dit, lorsqu'une opération normale est exécutée, le scénario ne peut continuer que lorsque cette opération est terminée et l'application client est donc bloquée également. Avec une opération nécessitant un long temps d'exécution, cela doit pouvoir se faire sans bloquer l'utilisateur, il est donc nécessaire de mettre en place un mécanisme qui va permettre l'exécution asynchrone d'une opération. Une telle opération se déroulera en deux étapes. La première étape consistera à demander l'exécution de l'opération, la seconde à récupérer les résultats produits. Entre ces deux étapes, le scénario sera bloqué mais il ne doit pas en être de même de l'application. Un tel mécanisme va intervenir à deux niveaux.

Premièrement, au niveau du scénario du service. En effet, le composant ayant en charge la gestion du scénario devra être capable de gérer une opération asynchrone et de permettre de reprendre le fil d'exécution une fois que les résultats de cette opération sont disponibles.

Deuxièmement, le fait de mettre en place un mécanisme permettant l'exécution d'opérations asynchrones implique de pouvoir mettre un service en « pause » le temps d'attendre que l'opération asynchrone soit terminée. Un composant doit donc être créé afin d'assurer la gestion de ces services « endormis ». Un service pourrait de la sorte être « figé » et cet état sauvegardé de façon à pouvoir reprendre ultérieurement.

Ce mécanisme pourrait également étendre ses fonctions à tous les services. Ainsi un utilisateur qui exécute un service particulièrement long (en terme de nombre d'étapes), pourrait s'interrompre et reprendre plus tard là où il en était. Il faut bien sûr, pour cela, que les services invoqués acceptent ce genre de manipulations.

Pour permettre la gestion des opérations asynchrones, il est également nécessaire de pouvoir, à tout moment, vérifier l'état d'une opération (en cours, terminée, ...). Un composant devra donc également être mis en place pour permettre à l'utilisateur de vérifier si une opération asynchrone est terminée et, le cas échéant, permettre la reprise du fil d'exécution du service. Concrè-

tement, si une opération est terminée, l'utilisateur pourrait demander à « réveiller » un service « endormi » et pourrait continuer à utiliser normalement ce service.

8.4 Conclusion

Dans ce chapitre les caractéristiques attendues d'un client pour le système BIGRE ont été concrétisées et des ébauches de solutions pour mettre en place une application répondant à ces caractéristiuqes ont ensuite été présentées. Dans le chapitre suivant nous verrons comment ces solutions ont été mises en place.

Chapitre 9

Conception

9.1 Introduction

Dans le chapitre précédent des exigences ont été décrites et des ébauches de solutions apportées. Ce chapitre à pour but de présenter l'architecture réalisée sur base de ces ébauches de solutions de façon obtenir une architecture y répondant le mieux possible.

9.2 Diagrammes de classes

L'achitecture présentée ci-dessous a été construite de façon à ce que chaque « besoin » évoqué dans le chapitre précédent fasse l'objet d'un composant distinct. Bien isoler et séparer les composants est le fruit d'une volonté de rendre le système facilement modifiable. Les principaux traits de l'architecture concèrnent la mise en place du modèle de conception des fabriques abstraites pour la création des interfaces graphiques.

La figure 9.1 présente le diagramme de classes de l'architecture. Ce diagramme présente les différents composants imaginés pour répondre aux besoins isolés dans le chapitre précédent. En voici la description :

Client_Main : Ce composant représente le point d'entrée du système. Il symbolise l'interface globale avec laquelle interagit l'utilisateur.

LocalServiceManager : Ce composant se charge de la gestion de services en local. Un des besoins mis en évidence dans le chapitre précédent concerne l'éventualité d'un long temps d'exécution pour un service nécessitant la mise en place d'un moyen d'exécuter des opérations asynchrones. Pour ce faire, il est nécessaire de conserver le service dans l'état dans lequel il est au moment de l'appel de l'opération en question. Il faut donc prévoir un mécanisme pour pouvoir mettre ce service en

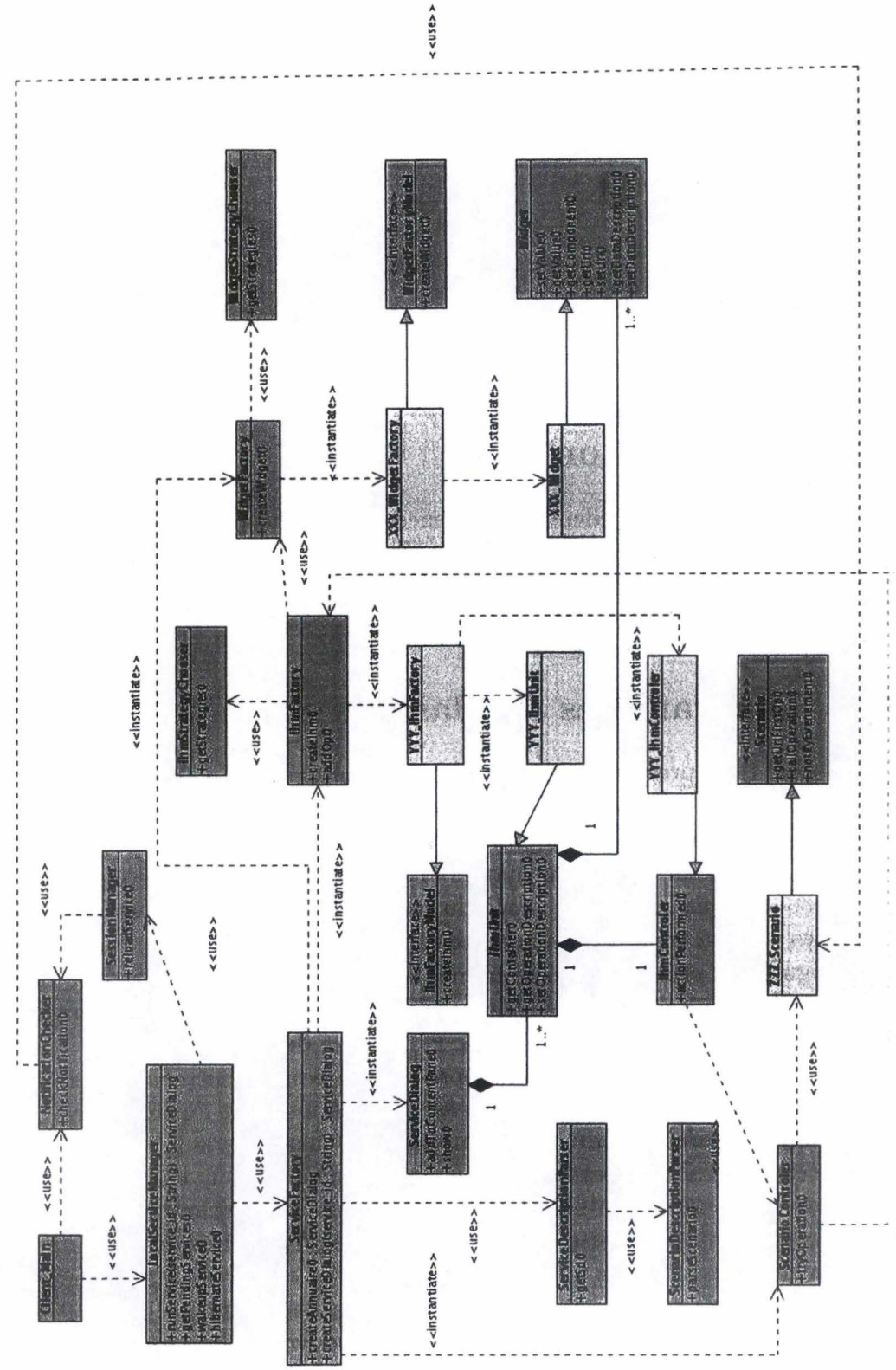

FIG. 9.1 – Architecture générique du système

« hibernation », et le « réveiller » lorsque l'opération est terminée. Il faut également prévoir un mécanisme permettant de contrôler si une opération est terminée ou non. Les composants ***SessionManager*** et ***NotificationChecker*** s'en chargent respectivement. Le rôle du ***LocalServiceManager*** consiste à maintenir la liste des services en « hibernation » et à gérer les actions du client concernant les services, à savoir la demande de réveil d'un service ou la demande d'exécution d'un service. Ce composant agit comme un dispatcher qui va appeler les composants nécessaires aux actions demandées.

SessionManager : Ce composant permet de placer un service en « hibernation » ou de « réveiller » un service « endormi ».

NotificationChecker : Ce composant permet de vérifier l'état d'une opération asynchrone.

ServiceFactory : Ce composant « amorce » la création de l'interface utilisateur d'un service. Il crée les différentes fabriques, instancie le contrôleur de scénarios et crée le composant qui représentera le service. La service factory est invoquée dès lors que le client demande à exécuter un service.

ServiceDescriptionParser : Ce composant à en charge l'analyse de la description d'un service.

ScenarioDescriptionParser : Ce composant se charge de l'analyse du scénario d'un service.

ScenarioControler : Ce composant représente le « cerveau » du système. Il a en charge la coordination des différents composants responsables de la création des interfaces ainsi que des composants prenant en charge les interactions avec l'interface. Le but de ce composant est de présenter à l'utilisateur une interface graphique construite selon ses préférences et présentant la ou les différentes opérations du service qu'il peut réaliser à un stade donné de l'exécution du service. Comme cela a été expliqué au point 8.3.1, un service est constitué d'un ensemble d'opérations atomiques coordonnées entre elles par un scénario. Ce scénario peut être vu comme le workflow du service, workflow dont les différents composants sont les opérations du service. Le ***ScenarioControler*** se charge donc d'assurer l'exécution de ce workflow afin que le service soit correctement exécuté. Pour effectuer le contrôle de l'exécution du scénario, ce composant se repose sur un moteur de scénario.

Scenario : Cette interface présente les fonctions qui doivent être implémentées par les moteurs de scénarios. Par moteur de scénarios, il est ici question, de façon simplifiée, d'un composant ayant en charge de préciser si une opération est exécutable ou non en fonction de l'état

actuel du déroulement du scénario.

ZZZ_Scenario : Ce composant représente un moteur de scénarios générique. Il doit se conformer à l'interface ***Scenario*** mais peut représenter n'importe quel type de moteur de scénarios. Ce choix permet de ne pas figer le moteur de scénarios dans l'architecture mais au contraire de le rendre facilement modifiable. Ce composant est crucial, car c'est sur lui que va reposer toute la gestion du déroulement du service. C'est lui qui va déterminer quelles sont les opérations exécutables à un stade donné du service mais également, selon la façon dont il sera implémenté, une possibilité d'annulation, une gestion de l'historique des opérations exécutées, etc.

Widget : Ce composant représente une classe abstraite qui définit les caractéristiques d'un widget. Un widget étant le plus petit élément graphique d'une interface.

WidgetStartegyChooser : Ce composant permet d'adapter le type de widgets en fonction des préférences de l'utilisateur.

WidgetFactory : Ce composant est une fabrique abstraite qui a en charge la création de widgets. En fonction de la stratégie définie par l'utilisateur grâce au ***WidgetStrategyChooser***, ce composant instancie une fabrique concrète qui aura en charge la création effective du widget.

WidgetFactoryModel : Cette interface définit le modèle que doivent adopter les fabriques concrètes de widgets.

XXX_WidgetFactory : Ce composant représente une fabrique concrète générique de widgets. Son but est de construire les widgets.

XXX_Widget : Ce composant représente un widget générique. Il représente un objet instancié par le composant ***XXX_WidgetFactory***. Ces deux composants sont représentés de façon générique afin de ne pas figer les fabriques de widgets dans l'architecture, laissant ainsi la possibilité d'ajouter des fabriques au système, du moment que celles-ci se conforment aux interfaces auxquelles elles se raportent (***WidgetFactoryModel*** et ***Widget***).

IhmStrategyChooser : Ce composant permet à l'utilisateur d'adapter le style de l'interface (formulaire, assistant, ...) à ses préférences.

IhmUnit : Ce composant représente une classe abstraite qui définit les caractéristiques d'une unité d'ihm. Une unité d'ihm peut être définie comme le pendant graphique d'une opération d'un scénario, c'est à dire un ensemble de widgets représentant une opération du scénario.

IhmControler : Ce composant représente une classe abstraite qui définit le mécanisme de contrôle d'une ***IhmUnit***. Ce mécanisme permet de rendre l'interface « réactive » en capturant les actions effectuées et

en les répercutant sur le ***ScenarioControler***.

IhmFactory : Ce composant représente une fabrique abstraite. Cette fabrique abstraite a en charge la création des unités ihm. En fonction des préférences définies par l'utilisateur, ce composant instancie une fabrique concrète qui aura en charge la création effective de l'unité ihm.

YYY_IhmFactory : Ce composant modélise une fabrique concrète générique d'unités ihm. Son but est de construire des objets du type YYY_IhmUnit.

YYY_IhmUnit : Ce composant représente une unité ihm générique. Il va de paire avec le composant ***YYY_IhmFactory*** qui l'instancie. Ces deux composants sont représentés de façon générique afin de ne pas les figer dans l'architecture de façon à pouvoir facilement ajouter de nouveaux types de fabriques d'unités ihm, du moment qu'elles se conforment aux interfaces définies.

YYY_IhmControler : Ce composant représente un controleur ihm générique et est lié aux composants ***YYY_IhmFactory*** et ***YYY_IhmUnit***. Il représente de façon générique le composant qui aura en charge la gestion des unités ihm créées par la fabrique concrète.

Grâce au mécanisme des fabriques abstraites l'application est facilement modifiable pour ajouter de nouveaux types de widgets ou de nouveaux types d'ihm. La figure 9.2 représente un exemple d'architecture où les composants génériques ont été remplacées par des composants concrèts.

Le composant *PetriNetwork* remplace le composant *ZZZ_Scenario* et représente un moteur de scénario basé sur les réseaux de Pétri. Les composants *RemoteWidget* et *RemoteWidgetFactory* représentent respectivement des Widgets distants et la fabrique concrète qui les instancies. Des widgets distants sont des widgets dont les classes d'implémentations ne sont pas disponibles en local. La fabrique doit donc commencer par rapatrier la classe d'implémentation du widget, ce qu'elle fait au moyen du composant *RemoteWidgetFetcher* avant de pouvoir l'instancier. Les composants *LocalWidget* et *LocalWidgetFactory* représentent, quant à eux, respectivement des Widgets locaux (dont les classes d'implémentation sont disponibles localement) et la fabrique qui les instancie. Le composant *WidgetMapper* a pour vocation d'établir la correspondance entre un élément de la description d'un service et un widget local (par exemple un paramètre en entrée serait converti en champ d'édition). Enfin, les composants *FormIhmFactory*, *FormIhmUnit* et *FormIhmControler* représentent les composants nécessaires pour créer des interfaces du type « formulaire ».

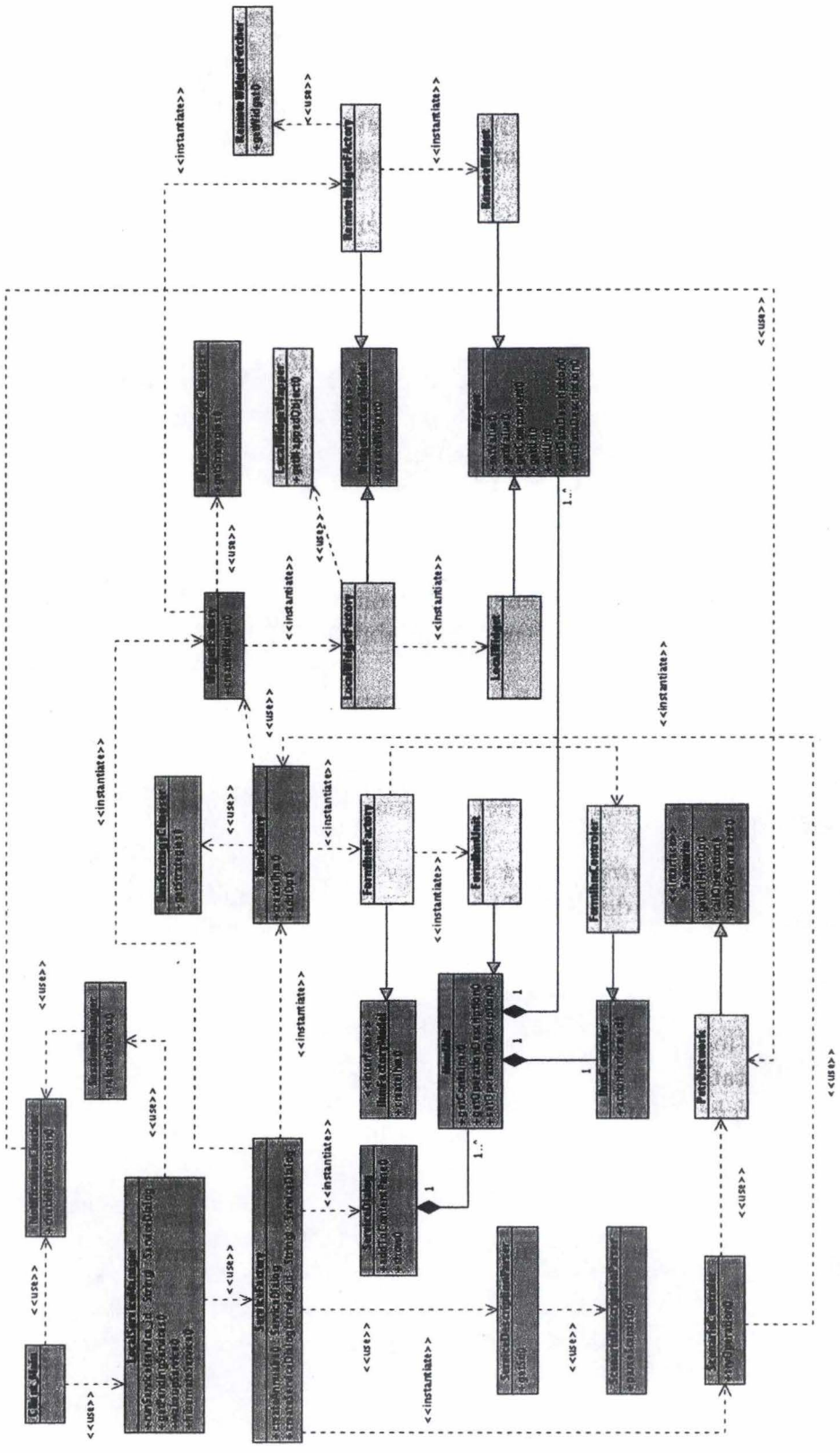

FIG. 9.2 – Architecture du système avec composants concrets

9.3 Diagrammes de séquence

Afin d'illustrer le fonctionnement de l'architecture décrite dans la section précédente, voici trois diagrammes de séquences basés sur l'architecture exemple de la figure 9.2. Le diagramme de la figure 9.3 illustre les étapes de la création de l'interface d'un service, le diagramme de la figure 9.4 illustre l'exécution d'une opération et le diagramme de la figure 9.4 illustre le « réveil » d'un service en attente de l'exécution d'une opération asynchrone.

La création de l'interface d'un service se déroule comme suit :

1. L'utilisateur demande l'exécution d'un service
2. Le composant *Client_Main* invoque le composant *LocalServiceManager* et lui demande de créer le service demandé.
3. Le composant *LocalServiceManager* invoque le composant *ServiceFactory* et demande la création de l'interface pour le service demandé.
4. Le composant *ServiceFactory* invoque le composant *ServiceDescriptionParser* pour qu'il effectue l'analyse de la description du service et qu'il structure les informations nécessaires à la création des interfaces du service.
5. Le composant *ServiceDescriptionParser* invoque le composant *ScenarioDescriptionParser* afin qu'il analyse le scénario du service.
6. Une fois l'analyse du service effectuée, le composant *ServiceFactory* instancie les composants *IhmFactory*, *WidgetFactory* et *ScenarioControler* puis passe le relais à ce dernier.
7. Le composant *ScenarioControler* instantie un composant implémentant l'interface *Scenario*. Dans ce cas, il s'agit du composant *PetriNetwork*.
8. Le *ScenarioControler* interroge le composant *PetriNetwork* pour connaître la première opération du service, d'après le scénario.
9. Une fois la première opération connue, le *ScenarioControler* demande au composant *IhmFactory* de créer l'unité ihm pour cette opération.
10. La fabrique d'ihm interroge alors le composant *IhmStrategyChooser* pour connaître les préférences de l'utilisateur. Dans ce cas, il s'agit d'une présentation en formulaire.
11. La fabrique d'ihm instancie alors une fabrique concrète correspondant au choix de l'utilisateur.
12. Le composant *IhmFactory* invoque alors le composant *WidgetFactory* pour créer les widgets nécessaires. Dans ce diagramme, la création des widgets n'est pas détaillée mais elle fonctionne sur le même principe que la création de l'ihm, à savoir, la fabrique de widget interroge le

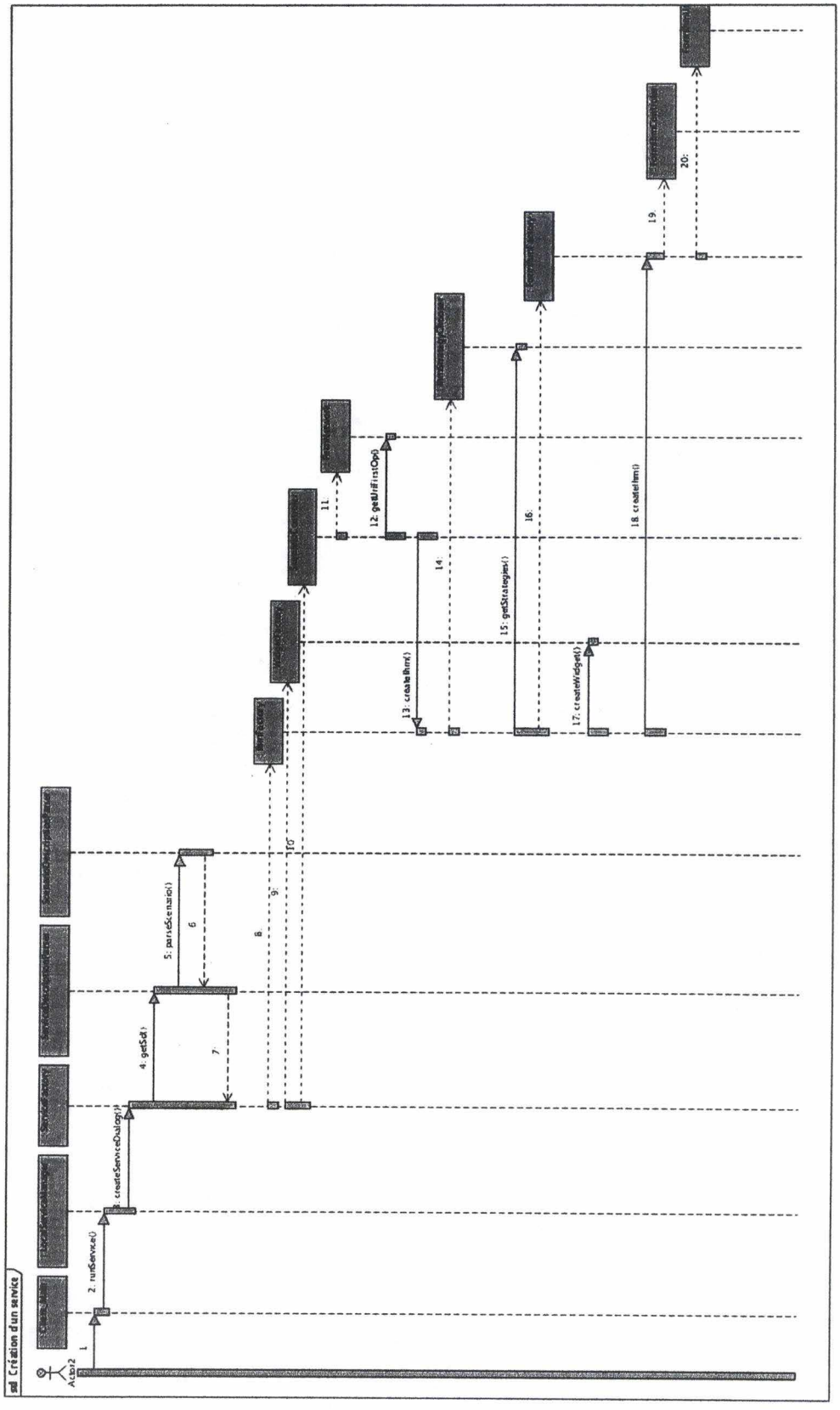

FIG. 9.3 – Création de l'interface d'un service

composant *WidgetStrategyChooser* pour connaître les préférences de l'utilisateur puis, en fonction, instancie la fabrique concrète correpondante pour créer les widgets.

13. Une fois les widgets créés, ceux-ci sont passés à la fabrique concrète d'ihm (ici *FormIhmFactory*) qui instancie un contrôleur pour l'ihm (ici *FormIhmControler*) et l'assigne au widget représentant l'exécution de l'opération (typiquement, un bouton). Ensuite, la fabrique concrète assemble les widgets et crée une unité ihm représentant l'opération (ici *FormIhmUnit*).
14. Cette unité ihm ainsi créé est alors présentée à l'utilisateur qui peut commencer à exécuter le service.

Comme l'illustre le diagramme de la figure 9.3, le composant *ScenarioControler* occupe une place cruciale au sein de l'architecture. Il assure le rôle de coordinateur pour les différents composants pendant toute la période de création et d'utilisation de l'interface. Cette position cruciale est issue de l'interprétation qui a été faite de la description d'un service. Nous avons en effet interprété la description d'un service comme la description d'un workflow. Les différentes opérations du service constituent les composants du workflow et le scénario représente le mode d'emploi de ce workflow, à savoir l'ordre d'exécution des opérations. Cette interprétation nous a poussé à considérer le scénario comme le « cerveau » du service et c'est donc naturellement que le composant chargé de veiller à la bonne exécution de ce scénario s'est retrouvé à une place prépondérante dans l'architecture.

Dans l'architecture exemple illustrée à la figure 9.2 et dans le diagramme de séquences illustré à la figure 9.3 apparaît un composant *PetriNetowrk*. Il s'agit d'un moteur de scénarios. La conception de ce composant est expliquée à la section suivante.

L'exécution d'une opération se déroule de la façon suivante :

1. L'utilisateur demande l'exécution d'une opération d'un service
2. L'action est capturée par le contrôleur de l'ihm (ici *FormIhmControler*) qui interroge le composant *ScenarioControler* pour savoir si l'opération est réalisable.
3. Le *ScenarioControler* interroge alors le moteur de scenarios (dans ce cas : *PetriNetwork*) en lui signalant l'opération que l'utilisateur tente d'exécuter. Le moteur de scénarios exécute l'opération (si c'est faisable) et retourne la liste des opérations qui peuvent alors être exécutées.
4. Le *ScenarioControler* invoque alors la fabrique d'ihm pour qu'elle construise les unités ihm nécessaires pour représenter les opérations suivantes.

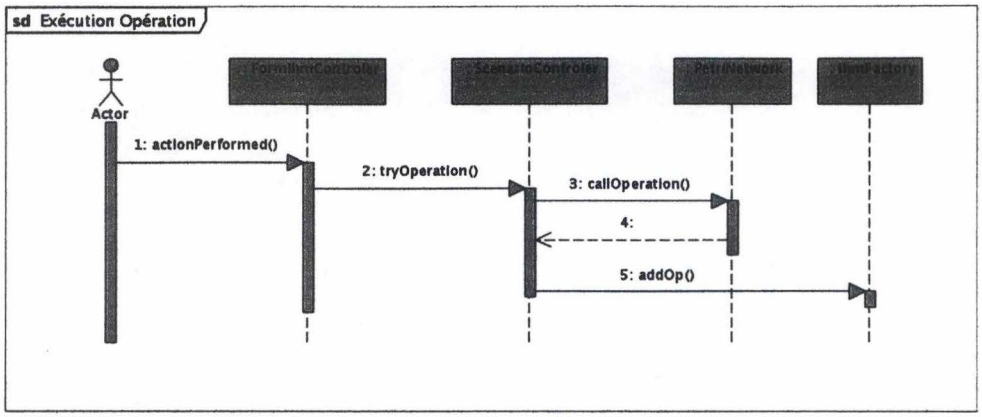

FIG. 9.4 – Exécution d'une opération

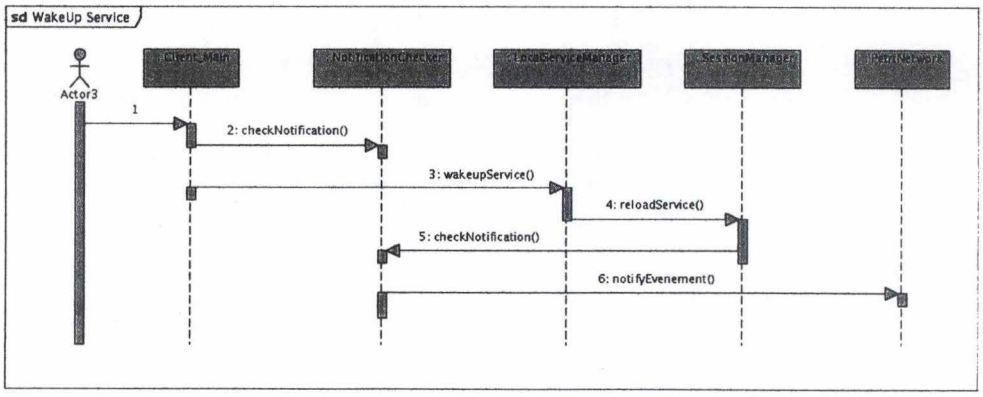

FIG. 9.5 – « Réveil » d'un service

Un service en attente d'une opération asynchrone est mis en « hibernation » en attendant que l'opération soit exécutée. Voici ce qu'il se passe lorsque l'opération est terminée :

1. L'utilisateur demande de vérifier si l'opération asynchrone dont il attend l'exécution est terminée. Le composant *NotificationChecker* se charge de cette vérification.
2. L'utilisateur apprend que l'opération est terminée. Il décide alors de poursuivre l'exécution du service en attente de cette opération. Via la composant *LocalServiceManager*, il peut demander le « réveil » de ce service.
3. Le *LocalServiceManager* invoque alors le *SessionManager* pour recharger le service.
4. Le *SessionManager* invoque à nouveau le *NotificationChecker* qui va alors signaler au moteur de scénarios que l'opération est bien terminée. L'utilisateur pourra alors reprendre le service là où il en était.

9.4 Moteur de scénarios

Comme évoqué dans la section 9.2, et comme illustré dans les diagrammes de séquences de la section précédente, le moteur de scénarios constitue un point crucial de l'application. C'est lui qui assure le bon déroulement de l'exécution du service. Son implémentation est cependant laissée libre dans l'architecture de façon à ce qu'il puisse être facilement modifiable.

Pour le développement du prototype, nous nous sommes appuyés sur le formalisme des réseaux de Petri pour réaliser un moteur de scénarios. Nous avons opté pour ce formalisme de façon arbitraire parce qu'il s'agissait du formalisme que nous connaissions le mieux. Un autre formalisme, tel que les machines à états aurait cependant pu parfaitement convenir.

Deux étapes ont été nécessaires pour réaliser un moteur de scénarios basé sur le formalisme des réseaux de Petri. La première étape à consisté en la réalisation d'une architecture capable de modéliser un réseau de Petri, mais surtout d'en contrôler le déroulement. La figure 9.11 illustre cette architecture. La seconde étape consistait en l'élaboration d'un langage de définition de scénarios permettant de modéliser un scénario comme un réseau de Petri. Nous allons maintenant expliquer comment modéliser un scénario au moyen d'un réseau de Petri ainsi que le modèle de scénarios mis au point. Ensuite nous présenterons l'architecture mise en place pour la modélisation et le contrôle de ces réseaux.

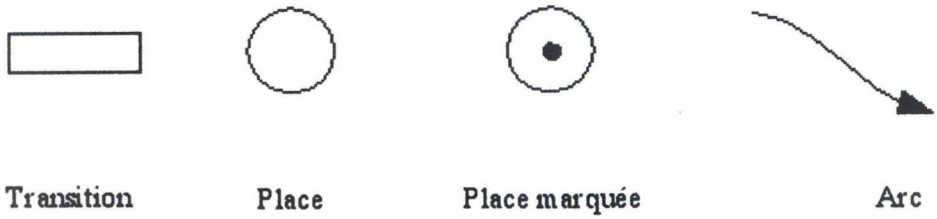

FIG. 9.6 – Concepts de base d'un réseau de Petri

9.4.1 Modélisation d'un scénario et représentation au moyen d'un réseau de Petri

Les explications concernant les réseaux de Petri sont issues de la référence suivante : [RDP]

Le formalisme des réseaux de Petri a été mis au point par Carl Adam Petri en 1964 [Wik]. Il permet, entre autre, de modéliser des systèmes dynamiques et d'étudier leur comportement.

Les deux concepts principaux des réseaux de Petri sont les places et les transitions. Une place modélise une condition du système (ou une ressource) et les transitions représentent les actions qui se déroulent dans le système et dont la réalisation dépend de l'état de ce dernier. L'état du système étant connu en observant l'ensemble des places.

Pour signifier qu'une condition est remplie, on place une « marque » dans la place. Cette marque est représentée par un jeton. Les places et transitions sont reliées entre elles par des arcs. Lorsqu'un arc relie une place à une transition, cela signifie que pour que cette transition puisse être « tirée »(une action du système exécutée), il faut que la condition soit remplie, c'est à dire que la place contienne une marque. Inversément, lorsqu'un arc relie une transition à une place, cela signifie que, une fois la transition « tirée », cette place contiendra une marque.

Le marquage initial d'un réseau représente les places marquées à l'initialisation du système. Ce marquage évolue ensuite en fonction des transitions franchies et donne, à tout instant, l'état du système.

Un scénario peut être défini comme l'ordonnancement d'une succession d'étapes. Dans le cadre d'un service bioinformatique, cela peut être ramené à la description de l'ordre d'exécution d'une suite d'opérations. Modéliser un tel scénario revient à exprimer cet enchaînement d'opérations. Chaque opération étant en fait conditionnée par le fait que les opérations qui la précèdent sont

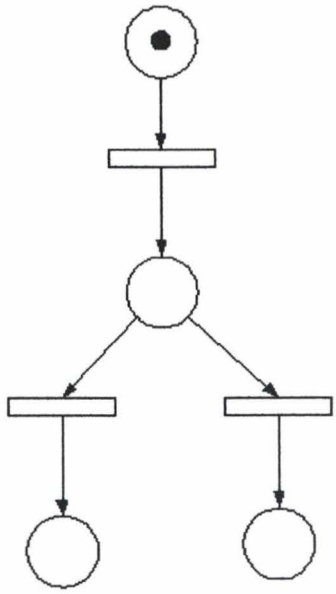

FIG. 9.7 – Exemple de scénario modélisé avec un réseau de Petri

effectivement terminées.

Un scénario peut donc être décrit par une opération de départ, point d'entrée du scénario. Puis, pour chaque opération, la liste des opérations qui pourront être exécutées lorsqu'elle sera terminée.

Dans le formalisme des réseaux de Petri, une opération pourrait être modélisée par une transition, et l'état d'une opération par une place de sortie. Cette place de sortie contiendrait un jeton lorsque l'opération représentée par la transition serait effectivement terminée. Les opérations suivantes auraient alors cette place comme place d'entrée. Pour représenter l'opération de départ, une place supplémentaire serait créée en aval de la transition modélisant cette opération et le marquage initial du réseau placerait un jeton dans cette place.

La figure 9.7 illustre un scénario modélisé par un réseau de Petri selon le formalisme décrit ci-dessus. Le marquage initial permet le franchissement de la première transition. Une fois cette transition franchie, la place de sortie est marquée permettant de franchir l'une des deux transitions restantes.

Ce petit exemple illustre rapidement le problème rencontré en se contentant de décrire un scénario comme une succession d'opérations.

En effet, dans le cas présent, lorsqu'une opération est terminée, seule une des opérations déclarées comme ses suivantes peut être exécutée. De fait, dans l'exemple de la figure 9.7, lorsque la première transition est franchie,

seule une marque est déposée dans la place de sortie et donc seule une des deux transitions suivantes peut être franchie.

Le problème qui se pose est que lors de l'exécution d'un scénario une opération peut parfois être suivie de plusieurs autres, non mutuellement exclusives. Ca cas de figure peut facilement être représenté au moyen d'un réseau de Petri en créant autant de places de sorties qu'il n'y a d'opérations non mutuellement exclusives.

Par exemple, imaginons une opération O1 qui peut éventuellement être suivie d'une opération O2, O3 ou O4 sachant que O2 et O3 sont mutuellement exclusives, c'est à dire que soit O2 soit O3 peut être exécutée mais pas les deux. Cela se traduirait par le réseau de Petri illustré à la figure 9.8.

Il est dès lors nécessaire que le formalisme décrivant le scénario permette de préciser quelles sont les opérations mutuellement exclusives et quelles sont les opérations qui ne le sont pas.

Pour ce faire, on peut décrire un scénario comme une suite de groupes d'opérations mutuellement exclusives, ces groupes n'étant eux pas mutuellement exclusifs. Ce mécanisme permet ainsi de représenter les deux cas décrits mais permet en plus de rendre certaines opérations mutuellement exclusives avec certaines autres sans l'être avec toutes.

Ce concept de « groupes d'opérations » reste cependant propre au formalisme de description du scénario et ne possède pas de représentation propre dans le résau de Petri. Il permet s'implement de grouper sous la même place d'entrée les opérations mutuellement exclusives.

La figure 9.8 illustre un exemple de réseau modélisé selon se principe. Les ellipses pointillées illustrent les groupes non exclusifs mutuellement d'opérations mutuellement exclusives. Dans ce cas de figure, les deux opérations reprises dans l'ellipse de gauche ne peuvent toutes les deux être exécutées. Par contre, l'opération comprise dans l'ellipse de droite peut être exécutée indépendamment de l'opération de l'ellipse de gauche qui aura été choisie.

En procédant de la sorte survient un problème supplémentaire généré par l'apparition du paralèllisme dans l'exécution des branches d'un scénario. En effet, si on reprend l'exemple de la figure 9.8 et que l'on imagine que chacune des places du bas désigne un état terminal du service, c'est à dire une opération qui clôture le service, autrement dit, une opération après laquelle il n'est plus possible de faire quoi que ce soit, on constate que dans le formalisme actuel, rien ne permet de préciser une telle opération, et donc, en repartant de l'exemple, on constate que bien que, par exemple, la place du bas de l'ellipse de droite soit marquée, rien n'empêche d'exécuter l'une ou l'autre des opérations de l'ellipse de gauche (pour autant qu'un jeton soit toujours présent dans leur place d'origine). Il faut donc également que le formalisme de description de scénario puisse spécifier qu'une opération est terminale. De

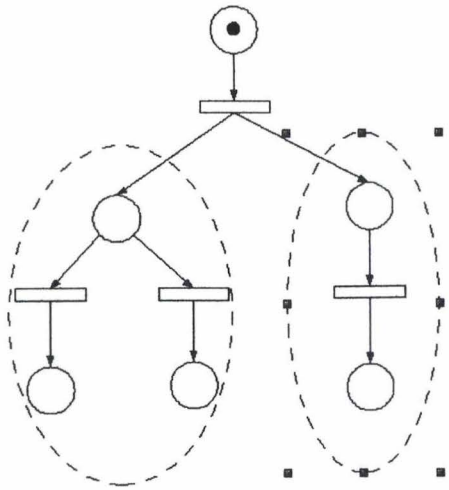

FIG. 9.8 – Exemple de scénario modélisé avec un réseau de Petri - 2

nouveau, ce concept ne trouve pas de représentation directe dans le réseau de Petri mais sera directement utilisé par le moteur de scénarios pour savoir si un scénario est terminé ou non.

Le formalisme ainsi constitué permettrait de représenter n'importe quel scénario constitué d'un enchaînement d'opérations.

Ce formalisme se doit cependant d'être enrichi de quelques structures de contrôle afin de permettre la modélisation de scénarios élaborés.

Une première structure de contrôle concerne l'exécution d'une opération. En effet, celle-ci pourrait être conditionnée par un élément extérieur à l'enchaînement des opérations. Ainsi, une opération pourraît être conditionnée à la validité de ses paramètres.

Une deuxième structure de contrôle concerne la validité de l'exécution d'une opération. En effet, dans le formalisme actuel, dès qu'une opération est exécutée, les places en sortie sont marquées permettant ainsi de poursuivre le scénario. Or, imaginons le cas d'une opération d'identification dans un système. La validité de l'exécution de cette opération dépend en fait du résultat de l'identification par le système et non uniquement du fait que cette opération ait été exécutée. En effet, il est nécessaire de vérifier que la validité de l'identification est positive avant de permettre de continuer l'utilisation de l'application. Il faut donc mettre en place un moyen permettant de conditionner le marquage d'une place. Cette condition peut être placée sur l'arc reliant la transition à la place. En procédant de la sorte, l'exécution de l'opération n'est pas entravée mais la suite du scénario dépend du résultat de cette opération. La figure 9.9 illustre ce concept.

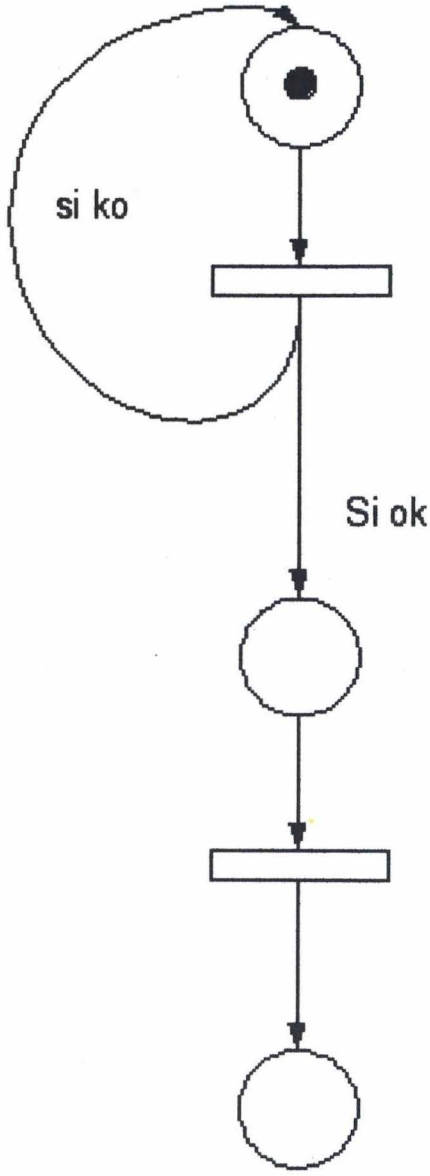

FIG. 9.9 – Exemple de scénario modélisé avec un réseau de Petri - 3

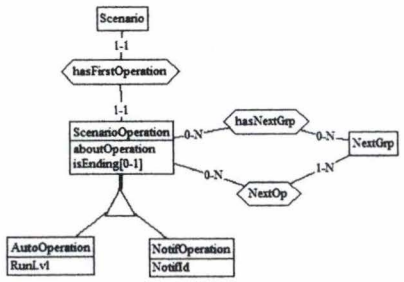

FIG. 9.10 – Modélisation d'un scénario

Une troisième structure de contrôle concerne ce que l'on pourrait appeler le mode d'exécution du scénario. Par exemple un scénario pourrait proposer deux modes d'exécution, un mode normal et un mode expert. Ce scénario serait composé de 4 opérations. Dans le mode normal seule une opération nécessiterait une intervention de l'utilisateur alors que dans le mode expert les 4 opérations demanderaient l'intervention de l'utilisateur. Il faut donc mettre au point un mécanisme capable d'exécuter de façon automatique certaines des opérations, en fonction du mode choisi.

Pour finir, un dernier mécanisme concerne la possibilité d'effectuer des opérations asynchrones comme cela a été décrit à la section 8.2. L'exécution d'une opération asynchrone entraîne le « blocage » du scénario (dumoins en ce qui concerne les opérations dépendant de l'opération asynchrone) jusqu'à ce que cette opération ait été exécutée. La ou les opérations qui suivent une opération asynchrone doivent donc attendre que cette opération ait fourni un résultat. Pour permettre cela, il suffit d'ajouter une place d'entrée aux opérations suivantes d'une opération asynchrone. Le marquage de cette place serait alors conditionné par l'exécution de l'opération attendue.

Sur base des besoins décrits ci-dessus, nous avons mis au point un formalisme permettant de modéliser un scénario. La figure 9.10 en présente la structure.

Scenario : Ce concept modélise le scénario. Un scénario à au moins un première opération.

ScenarioOperation : Ce concept modélise une opération du scénario. Elle concèrne une opération du service (*aboutOperation*) et peut être une opération finale (***isEnding***).

AutoOperation : Une AutoOperation constitue une opération qui peut être exécutée de façon automatique par le scénario en fonction du niveau utilisateur choisi et qui est spécifié par *RunLvl*. Si le *RunLvl* d'une opération est supérieur au niveau utilisateur choisi, alors l'opération est

exécutée de façon automatique.

NotifOperation : Ce concept modélise une opération asynchrone. Son attribut *NotifId* permet d'identifier la place supplémentaire créée pour permettre le blocage du scénario comme cela a été décrit ci-dessus.

NextGrp : Ce concept modélise un groupe d'opérations suivantes. Il est lui même constitués d'une d'opérations. Les différentes opérations qui constituent un groupe sont mutuellement exclusives mais les groupes ne sont pas mutuellement exclusifs entre-eux. L'attribut *hasCondition* permet de conditionner le marquage de la place permettant l'exécution des opérations de ce groupe.

L'annexe A contient la description en OWL de ce modèle.

A l'aide de ce modèle, il est possible de décrire le scénario d'un service et de le transformer en un réseau de Petri selon les règles suivantes :

1. Une *ScenarioOperation* représente une transition. S'il s'agit de la première opération, elle est associée à une place d'entrée initialement marquée.
2. Pour chaque groupe d'opérations (*NextGrp*) d'une opération, une place de sortie est créée. Chacune des opérations de ce groupe se voit alors attribuer cette place comme place d'entrée. Si une des opérations de ce groupe est une opération notifiable, elle se voit attribuer une place d'entrée supplémentaire. Si un groupe est frappé d'une condition, l'arc reliant l'opération précédente à la place d'entrée de ce groupe se voit associé à une condition dont dépendra le marquage de cette place.

Pour assurer l'exécution correcte de ce réseau et la prise en compte des paramètres supplémentaires (opération automatique, etc.) un moteur spécial a été conçu et est expliqué dans la section suivante.

9.4.2 Architecture Moteur de réseaux de Petri

La figure 9.11 présente l'architecture créée pour modéliser un scénario s'appuyant sur le formalisme décrit à la section précédente. Le but de cette architecture est de permette la modélisation et le contrôle de l'exécution de ce scénario. Le formalisme modélisé se limite à l'utilisation des places commes des conditions booléennes et non comme des ressources (comme l'autorise le formalisme des réseaux de Petri). Cette limitation ne se révèle cependant pas contraignante pour l'utilisation qui doit être faite du modèle, à savoir l'exécution d'un scénario. Le fil d'exécution d'un scénario étant conditionné par le fait qu'une (ou plusieurs) opération soit terminée ou non.

Voici la description du modèle :

Place : Ce composant modélise une place du réseau. Une place peut contenir un *Token*.

Token : Ce composant modélise un jeton du réseau.

Transition : Ce composant modélise une transition du réseau. Il est associé à une opération du service dont le scénario est modélisé.

ArcOut : Ce composant modélise un arc sortant d'une place et entrant pour une transition. La place constitue l'origine de l'arc et la transition la destination.

ArcIn : Ce composant modélise un arc ayant pour origine une transition et pour destination une place.

ConditionArc : Cette interface modélise une condition sur un arc. Elle permet de refuser le marquage de la place de destination si la condition précisée n'est pas remplie.

ConditionTir : Cette interface modélise le concept de gardien sur une transition. Il empêche l'exécution de la transition si la condition du gardien n'est pas remplie.

Marquage : Ce composant modélise le marquage courant du réseau.

MarquageInitial : Ce composant représente le marquage initial du réseau.

PetriNetwork : Ce composant assure la gestion du réseau (franchissement des transitions, mise à jour du marquage, etc.)

Historique : Ce composant permet une gestion de l'historique d'un scénario.

OperationHisto : Ce composant modélise une opération de l'historique. L'idée est que une fois qu'une transition est franchie, un enregistrement soit effectué de l'opération exécutée, des paramètres fournis, des résultats obtenus afin de pouvoir garder une trace de toutes ces informations. Cela pourrait par exemple permettre l'annulation de certaines étapes, la récupérations de résultats obtenus, etc. Avec le composant *Historique*, ce composant constitue la « mémoire » du scénario.

9.5 Modélisation du scénario par contraintes

Dans le chapitre précédent nous avons proposé un formalisme permettant de décrire un scénario comme un enchaînement d'opérations. Ce formalisme permet de décrire de façon très stricte le déroulement d'un scénario. Si ce formalisme semble parfaitement adapté pour décrire le scénario d'un service dont l'exécution est très rigide - nous entendons par là un service pour lequel l'ordonnancement des opérations est identique à chaque exécution - il se révèle beaucoup plus faible pour décrire le scénario d'un service très flexible. En

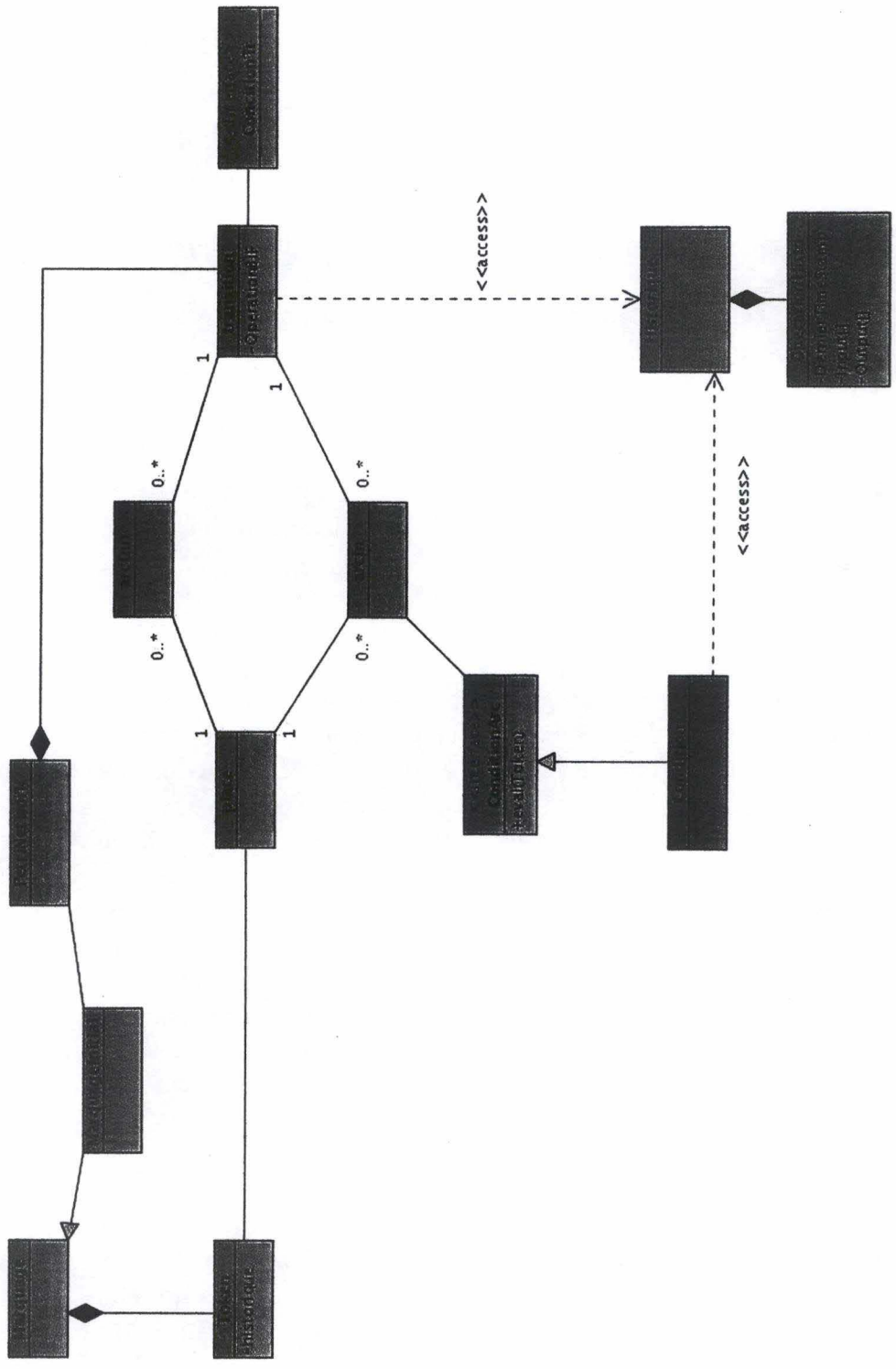

FIG. 9.11 – Moteur de réseaux de Petri

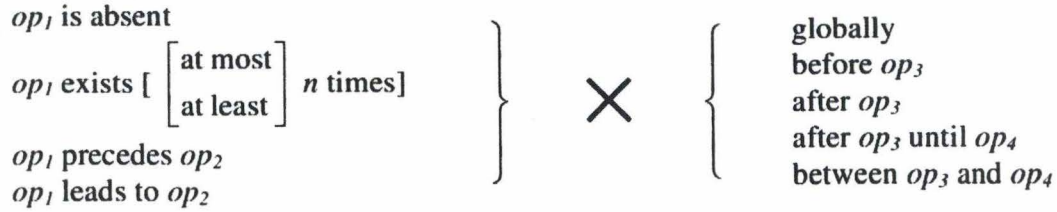

FIG. 9.12 – Modèle de règles

effet, si pour un même service plusieurs ordres d'exécution des opérations sont possibles, la description du scénario va rapidement se révéler très complexe.

L'article « *Pattern-Based Specification and Validation of Web Services Interaction Properties* » ([LHJ05]) propose une approche différente du contrôle de l'exécution d'un service. Les auteurs préconisent l'emploi d'un système de règles pour définir les propriétés d'interaction d'un service, c'est à dire la façon dont on peut interagir avec le service. Plutôt que de définir précisément le flux d'exécution d'un service comme on le ferait en décrivant un scénario d'exécution, ils proposent de spécifier à travers une série de règles, construites selon un certain modèle, les opérations qui peuvent, ou pas, être exécutées en fonction de l'état d'exécution du service.

Le modèle sur lequel sont basées les règles est illustré à la figure 9.12 et permet de définir l'existence et l'ordonnancement des opérations, ainsi que la portée des règles définies. La situation suivante : une opération O_2 peut être exécutée au maximum trois fois entre l'opération O_1 et l'opération O_3 serait ainsi modélisée par la règle « O_2 **exists at most 3 times after** O_1 **until** O_3 ».

Les auteurs définissent également une ontologie pour définir ces règles et proposent également un framework de validation de ces règles basé sur des machines à états. De façon simplifiée, le framework fonctionne de la façon suivante : lors de l'exécution d'un services, les messages entre l'utilisateur du service et le service lui-même sont interceptés et « analysés ». Si il existe une règle concernant le message intercepté, sont état est vérifié. Si toutes les règles se rapportant au message peuvent être appliquées, le message est considéré comme valide, sinon une erreur est signalée.

La définition des propriétés d'interaction d'un service par des règles plutôt que par la description d'un scénario d'exécution se révèle en fait beaucoup plus flexible. En effet, si, par exemple, une opération vient à être ajoutée à un service, la description du scénario de ce service risque de devoir être entièrement retravaillée. De plus, si le scénario est complexe, il sera d'autant plus difficile d'y intégrer la nouvelle opération. Lors de la définition des propriétés d'interaction d'un service par règles, il suffit d'ajouter ou de modifier une ou

plusieurs règles pour que l'opération soit intégrée au service. Le fait d'employer des règles pour définir les propriétés d'interaction d'un service permet également de donner plus de souplesse au service lui-même. En effet, tant que les règles définies sont respectées, rien n'empêche l'utilisateur d'utiliser le service d'une façon peut-être non-initalement prévue par le fournisseur du service.

Définir les propriétés d'interaction d'un service au moyen de règles offre donc une alternative intéressante à la définition d'un scénario et présente donc une piste intéressante que nous n'avons malheureusement pas pu explorer plus en avant par manque de temps.

9.6 Conclusion

Dans ce chapitre, l'architecture créée a été présentée et quelques exemples d'utilisation ont été expliqués. Cette architecture a été pensée dans une optique de modulabilité et d'adaptabilité afin de permettre d'implémenter un système très souple et facilement modifiable. La caractéristique principale de cette architecture réside dans l'utilisation du modèle de conception dit de « la fabrique abstraite » , couplé à un gestionnaire de stratégies, ce qui offre une grande souplesse et adaptabilité permettant de modifier dynamiquement le comportement du système.

De plus, l'architecture a été conçue de façon à pouvoir modifier facilement les points cruciaux du système (fabrique de widgets, fabrique d'ihm, moteur de scénario) sans que cela n'ait trop d'incidence sur le reste de l'application.

La création d'un formalisme pour décrire les scénarios ainsi que l'architecture d'un moteur de scénarios basé sur ce formalisme ont également été détaillée, ainsi qu'une piste alternative intéressante, selon nous, à explorer.

Dans le chapitre suivant, quelques détails de l'implémentation sont exposés ainsi qu'un exemple d'utilisation du prototype.

Chapitre 10

Implémentation

10.1 Introduction

Dans le chapitre précédent, l'architecture générale a été présentée. Ce chapitre présente le prototype qui a été réalisé pour valider cette architecture. Une première section définit les outils qui ont été employés pour réaliser ce prototype. Vient ensuite l'explication des choix d'implémentation qui ont été effectués ainsi que quelques détails concernant l'implémentation. Le chapitre est alors clôturé par une illustration du prototype par un exemple d'exécution d'un service.

10.2 Outils utilisés

Cette section présente les outils qui ont été utilisés pour réaliser le prototype et l'explication de ces choix.

10.2.1 Sun JAVA2 Standard Edition 1.4

Ce prototype a été réalisé en JAVA. Le langage Java est le langage utilisé dans le système BIGRE, notamment pour le framework d'invocation des services. Comme le client doit, à terme, utiliser ce framework, l'utilisation du langage Java s'est imposée d'elle-même. Néanmoins, étant donné que le langage Java est le langage orienté objet que nous maîtrisons le mieux et qu'il présente une grande souplesse au niveau de la réalisation des interfaces graphiques, il est fort probable que nous aurions effectué ce choix si la possibilité nous en avait été laissée. Java présente également l'avantage d'être un langage portable et possède un mécanisme d'introspection permettant le

chargement dynamique de classes, un mécanisme employé lors de la création des fabriques afin de les rendre plus dynamiques.

10.2.2 Jena

La description des services fournis par BIGRE ainsi que l'ontologie qui les définit sont réalisés en OWL. Etant donné que la réalisation de l'interface d'un service commence par l'analyse de la description de ce dernier, il a rapidement été nécessaire de manipuler des données OWL. *Jena* fournit, entre autre, une API pour manipuler des données OWL et s'est donc révélée indispensable dès le début de l'implémentation.

10.2.3 Eclipse et CVS

Le développement du prototype s'est fait au moyen de la plateforme de développement Eclipse[1] qui offre, entre autre, un environnement intégré de développement spécialement conçu pour JAVA et qui se révèle très pratique et fonctionnel (compilation, débogage, complétion automatique de code, etc.). Eclipse intègre également un module CVS (Concurrent Version System), permettant de gèrer un *repository* du code et des différentes versions qui s'est révélé très pratique.

10.3 Choix d'implémentation

Le but du prototype réalisé était de valider les points critiques de l'architecture présentée au chapitre précédent. Dans cette optique, seul un sous ensemble de l'architecture a été implémenté. Les éléments le plus remarquables de cette implémentation sont décrits dans cette section.

10.3.1 Fabriques abstraites

Un des éléments principaux de l'architecture repose sur la mise en place du modèle de conception de la « fabrique abstraite ». Deux fabriques abstraites ont été implémentées dans le prototype. Celle responsable de la création des Widgets et celle responsable de la création des IhmUnit. Afin de rendre plus modulable encore l'application au niveau de ces fabriques, le mécanisme d'introspection de Java a été utilisé afin de permettre un chargement dynamique des fabriques concrètes. Cela offre une très grande souplesse au niveau de

[1] `http://www.eclipse.org`

l'ajout de nouvelles fabriques dans le système. Il suffit en effet que la nouvelle fabrique se conforme à l'interface définissant les fabriques concrètes dans l'architecture et qu'elle soit ajoutée aux stratégies possibles pour pouvoir être intégrée dans l'application.

Voici le listing de la classe abstraite ayant en charge le chargement dynamique des classes concrètes :

```
package widgets;

import java.lang.reflect.Method;
import java.util.ArrayList;

import parsing.DataDescription;

import exceptions.NoWidgetException;

public class WidgetFactory {                                              10
    public Widget createWidget(DataDescription dataDescr) throws NoWidgetException {
        WidgetStrategyChooser wsc = new WidgetStrategyChooser();

    ArrayList strategies = wsc.getStrategies();

    short j = 0;

        Widget myWidget = null;
        while ((myWidget == null) && (j < strategies.size())) {
            try {                                                         20
                // Création dynamique de la factory en fonction de la
                // stratégie
                String packageName = "widgets";
                Class wf = Class.forName(packageName + "." + (String) strategies.get(j));
                WidgetFactoryModel wfm = (WidgetFactoryModel) wf.newInstance();

                // Appel dynamique de la méthode pour créer le widget
                Class[] p1 = new Class[] { DataDescription.class };
                Method cw = wf.getMethod("createWidget", p1);
                                                                          30
                Object[] param = new Object[]{dataDescr};
                myWidget = (Widget)cw.invoke(wfm, param);
            }catch (Exception e) {
                j++;
            }
        }

        if(myWidget != null){
            dataDescr.setWidget(myWidget);
            return myWidget;                                              40
        }else{
```

```
                    //Le widget n'a pas pu être créé
                    throw new NoWidgetException();
                }
            }
        }
```

Cette fabrique commence par interroger le gestionnaire de stratégies. Dans cette implémentation, la gestion des stratégies se fait de façon très simple et le gestionnaire se contente de retourner la liste des classes à instancier, ordonnée selon les préférences définies. Le choix de retourner une liste plutôt qu'une seule classe permet de retomber sur le deuxième choix si la classe correspondant au premier choix pose problème, et ainsi de suite. La fabrique tente ensuite de charger dynamiquement la première fabrique concrète de la liste et de l'invoquer pour construire le widget. Si cela est possible, elle retourne le widget ainsi créé, sinon, elle tente de faire pareil avec la deuxième fabrique concrète de la liste. Si aucune fabrique ne parvient à instancier le widget, la fabrique retourne alors une exception signalant que la création du widget est impossible.

Le mécanisme est appliqué de façon similaire à la fabrique abstraite ayant en charge la création des unités Ihm.

10.3.2 Moteur de scénario

Le prototype réalisé intègre une implémentation d'un moteur de scénarios basé sur le formalisme décrit au chapitre précédent. Etant donné l'objectif du prototype, seule une partie de l'architecture du moteur a été implémentée. Cette partie permet la gestion d'un scénario sans tenir compte de l'historique, ni des structures de contrôle que sont les gardiens (*ConditionTir*), les arcs conditionnés (*ConditionArc*), les opérations automatiques ou les opérations asynchrones.

10.3.3 Détails techniques

Le prototype implémenté est capable sur la description d'un service de générer intégralement une interface graphique et ce de façon dynamique. L'interface générée reste cependant simple mais est fonctionnelle. Le prototype est également capable d'assurer la correcte exécution du scénario respectant le formalisme décrit au point précédent. A ce stade, l'exécution du service par le prototype est simulée et aucun échange ne se fait encore avec le système BIGRE.

En chiffres, le prototype se compose de :

- 61 classes Java

- 4167 lignes de code

10.4 Exemple

L'exemple illustré ci-dessous repose sur le petit scénario suivant : un utilisateur peut se connecter à une service pour acheter ou louer des films. La première étape consiste à signaler qui il est. Une fois identifié il peut choisir un film et décider de le louer ou de l'acheter. Après avoir fait ce choix il lui est demandé de choisir son moyen de payement (carte ou facture). A tout instant après sa connexion, il peut décider de se déconnecter.

Voici comment décrire ce scénario selon notre formalisme :

- L'opération initiale consiste à s'identifier. Comme le mécanisme de marquage conditionnel n'est pas encore implémenté dans le prototype, cette identification revient juste à signaler qui l'on est.
- Après l'identification, trois choix sont possibles. On peut choisir de louer un film, d'acheter un film ou tout simplement de se déconnecter. Acheter et louer sont cependant deux opérations exclusives mutuellement. L'opération initiale sera donc suivie de deux groupes d'opérations suivantes. Un premier groupe contiendra les opérations « achat » et « location » tandis que le second groupe contiendra l'opération de déconnexion qui sera définie comme finale.
- Une fois le choix effectué entre achat et location, il reste à l'utilisateur à choisir son moyen de payement, et ça, quel que soit son choix précédent. Un nouveau choix peut être fait : payement par carte ou par facture. Ces deux opérations constitue donc un nouveau groupe duquel seront suivies les opérations d'achat et de location. Les opérations de payements sont finale.

Modélisé selon notre formalisme cela donne le réseau de Pétri illustré à la figure 10.1.

La transition *T1* représente l'opération initiale (identification). Les opérations *T2* et *T3* constitue un groupe et représentent les opérations de location et d'achat. Ce groupe permet donc de s'assurer que le réseau de Petri n'autorisera l'exécution que d'une seule de ces deux opérations. L'opération *T6* constitue également un groupe et représente l'opération de déconnexion. Enfin, les opérations *T4* et *T5* constitue également un groupe et représentent les opérations de payement par carte et par facture. Les opérations *T4*, *T5* et *T6* sont des opérations terminales.

L'exécution de ce scénario par le prototype est illustrée par les figures 10.2, 10.3, 10.4 et 10.5. La description OWL du service est disponnible à l'annexe B

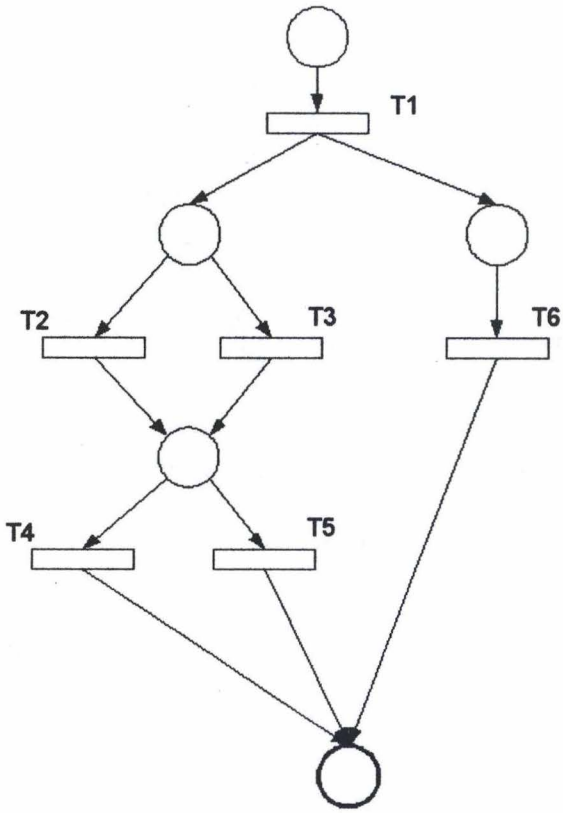

FIG. 10.1 – Exemple de scénario modélisé en réseau de Petri

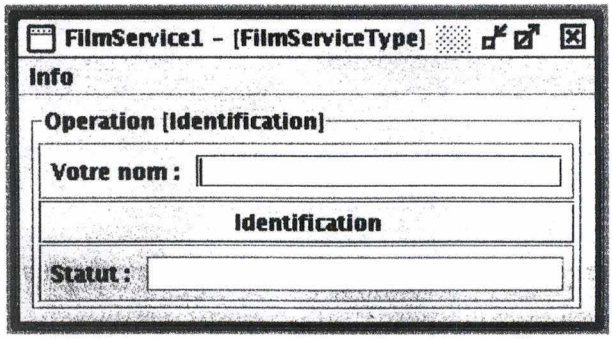

FIG. 10.2 – Exemple de scénario - Etape 1

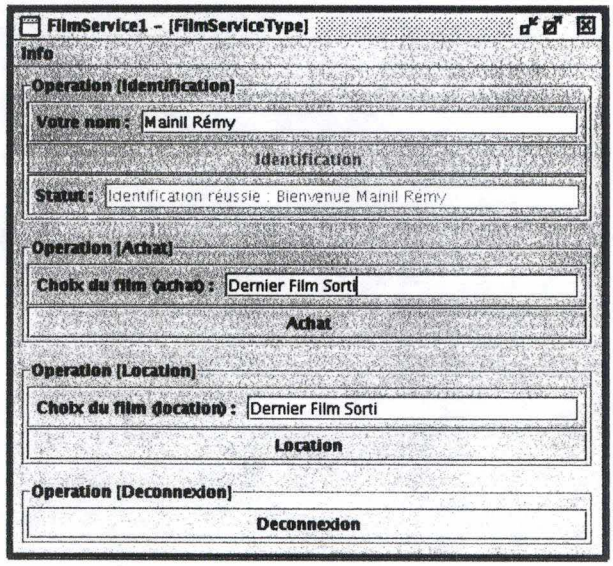

FIG. 10.3 – Exemple de scénario - Etape 2

La figure 10.2 présente l'opération définie comme initiale. Une fois exécutée, trois opérations sont possibles : achat, location et déconnexion. La figure 10.3 illustre ce cas. Si on clique sur l'opération « déconnexion » l'application s'arrête et plus aucune opération n'est exécutable. En l'état, dans le prototype, cela se traduit par le fait qu'un clic sur un bouton reste sans réponse et que le moteur de scénarios renvoie le message suivant :

```
(271) - Cette opération n'est pas exécutable !
```

La figure 10.4 illustre le cas ou on aurait choisit d'effectuer un achat et présente donc deux opérations supplémentaires tout en invalidant la possibilité de faire une location. Pour terminer, la figure 10.5 illustre le choix qui est effectué de payer par carte.

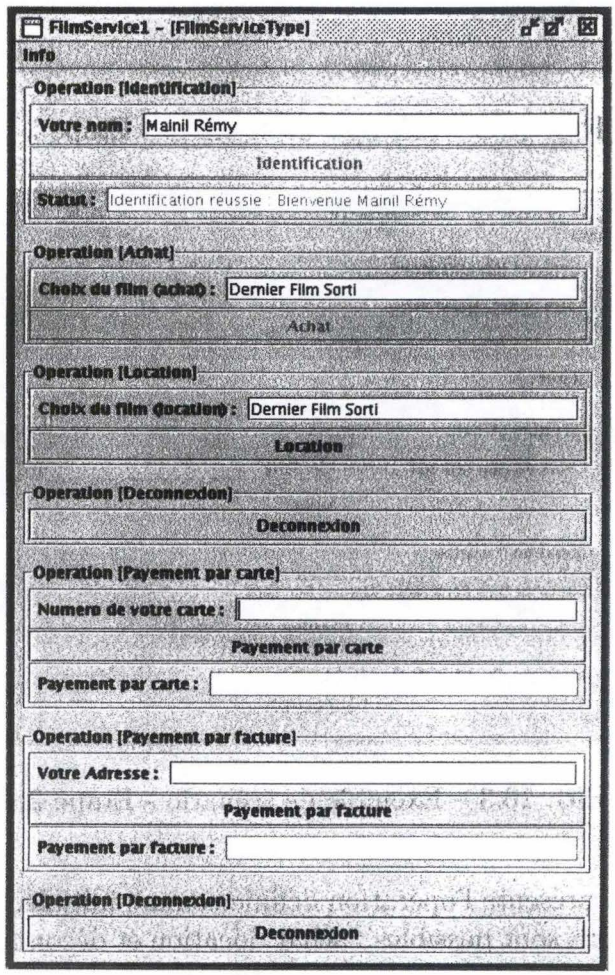

FIG. 10.4 – Exemple de scénario - Etape 3

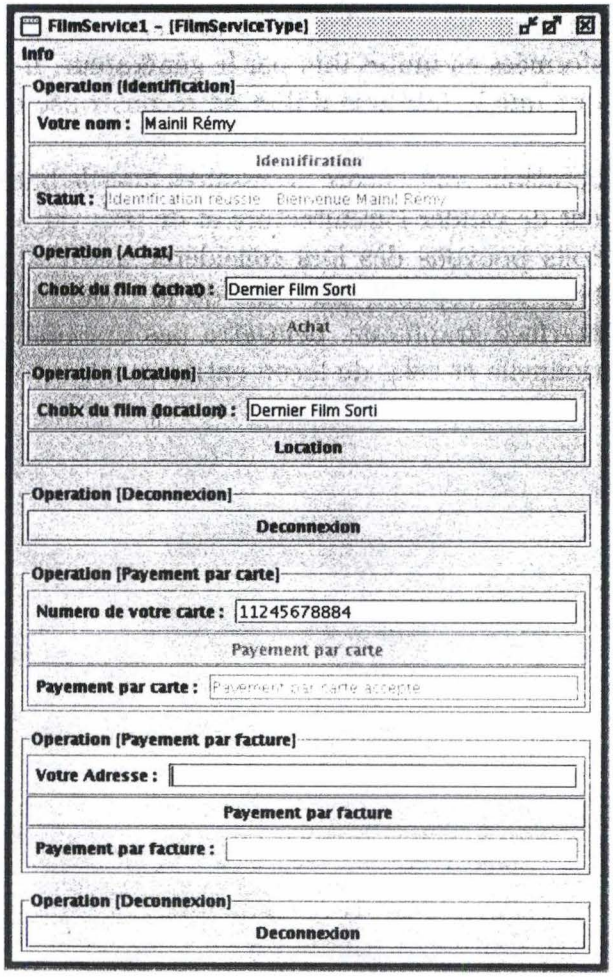

FIG. 10.5 – Exemple de scénario - Etape 4

10.5 Conclusion

L'exemple montré précédemment souffre de nombreux défauts. Ainsi, plutôt que de signaler par un message qu'une opération n'est pas réalisable il serait préférable d'en désactiver les composants (bouton et champs). On peut remarquer aussi que l'opération « Déconnexion » apparaît plusieurs fois. Cela vient du fait qu'actuellement, le moteur de scénario renvoie toutes les opérations exécutables à un moment donné du scénario. Toutes ces opérations sont ensuite transformées en unités ihm par le générateur. Il serait préférable et plus ergonomique que la fabrique d'ihm ne reconstruise que les nouvelles opérations.

D'autres améliorations sont encore nécessaires mais le but de ce prototype était principalement de valider l'architecture et de prouver la faisabilité d'un tel mécanisme. Nous pouvons dès lors considérer que l'objectif est rempli puisque le prototype est capable, sur base de la description d'un service, de générer une interface graphique, peut-être pas ergonomique dans l'état actuel, mais fonctionnelle et celà, de façon entièrement automatique.

Chapitre 11

Conclusion

La bioinformatique est une discipline en forte évolution. Les énormes progrès accomplis dans le domaine des télécommunication ces 20 dernières années ont permis de la rendre accessible à tous. Malheureusement, un problème persiste. L'accessibilité n'est pas au rendez-vous. En effet, de nombreux services et outils bioinformatiques ont été développés à travers le monde, indépendamment les uns des autres, en fonction des besoins du moment. Il en résulte que les services disponibles sont très hétérogènes et disparates rendant compliquée leur utilisation. Les ressources bioinformatiques souffrent également du même problème.

Le projet BIGRE est issu d'une volonté de simplifier l'utilisation de ces outils existants aux différents utilisateurs potentiels, mais également d'assurer une meilleure utilisation des ressources disponibles et de faciliter, à terme, la création de nouveaux outils. Les objectifs de ce projet sont donc d'apporter des solutions aux problèmes d'hétérogénéité et de disparités des outils et ressources bioinformatiques.

Pour résoudre le problème d'hétérogénéité des outils, la piste de réflexion proposée dans le mémoire de Pierre Buyle et Quentin Dallons [BD03] pronait la mise en place « *d'interface homme-machine intelligentes et dynamiques s'adaptant de manière automatique au service à utiliser* ». Ce mémoire s'est inscrit dans le cadre de la concrétisation de cette idée.

Après avoir défini les différents besoins d'un tel procédé nous avons mis en évidences les différentes étapes de la création dynamique de l'interface graphique d'un service. Sur base de ces étapes, nous avons réalisé une architecture capable de modéliser ce processus de création. Un prototype a ensuite été développé pour valider les points critiques et la faisabilité de l'architecture imaginée. Une attention particulière a également été portée sur la création d'un mécanisme capable d'assurer la coordination des différentes opérations proposées par un service.

Actuellement, le prototype développé est capable de générer, de façon entièrement dynamique et sur la seule base de la description d'un service. une interface graphique correpondant à l'utilisation du service.

Le mécanisme ainsi créé n'est pas encore utilisable tel quel pour exécuter un service. Nous estimons cependant qu'il peut constituer une base sur laquelle pourrait éventuellement s'appuyer le développement d'un client opérationnel pour la plateforme BIGRE.

Bibliographie

[BD03] Pierre Buyle and Quentin Dallons. Partage de ressources bioinformatiques hétérogènes - conception et implémentation d'une fédération de médiateurs. `http://www.info.fundp.ac.be/~ven/fichiers-memoire/2003-Buyle-Dallons.pd%f`, June 2003.

[CN03] Jean-Michel Claverie and Cedric Notredame. *Bioinformatics for Dummies.* Wiley Publishing, 2003. ISBN 0-7645-1696-5.

[Coa] The Workflow Management Coalition. The workflow management coalition. `http://www.wfmc.org` (Dernière mise à jour : 28/08/2006) (Dernière visite : 29/03/2006).

[DBDM04] Quentin Dallons, Pierre Buyle, Olivier Dugas, and Joseph Mavor. Bigre : Rapport scientifique. October 2004.

[Den04] Marie-Laetitia Denayer. Proposition de modélisation des services bioinformatiques dans le cadre d'une architecture fédérée. `http://www.info.fundp.ac.be/~ven/fichiers-memoire/2004%20-%20Denayer-Ma%rie-Laeticia.pdf`, June 2004.

[GHJV99] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, 1999. ISBN 2-1771-8644-7.

[Inf] Centre national de ressources informatiques appliquées à la génomique. `http://www.infobiogen.fr` (Dernière mise à jour : 31/01/2006) (Dernière visite : 17/04/2006).

[LHJ05] Zheng Li, Jun Han, and Yan Jin. Pattern-based specification and validation of web services interaction properties. In *ICSOC*, pages 73–86, 2005.

[RDP] Médi@tice - le formalisme des réseaux de pétri. `http://www.cyber.uhp-nancy.fr/demos/GEII-010/cha_2/cours_2_1.html` (Dernière mise à jour : 20/04/2006)(Dernière visite : 24/04/2006).

[Tava] Taverna. `taverna.sourceforge.net` (Dernière mise à jour :07/06/2006) (Dernière visite : 21/04/2006).

[Tavb] Taverna. An introduction to taverna 1.3. URL http://taverna.sourceforge.net/index.php?doc=docroot.html. (Basic Introductory Tutorial).

[Wik] Wikipedia. Wikipedia (fr). URL http://fr.wikipedia.org. http://fr.wikipedia.org (Dernière mise à jour : 30/08/2006) (Dernière visite : 15/04/2006).

Annexe A

Ontologie du modèle de scénario

```
<?xml version="1.0"?>

<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:owl="http://www.w3.org/2002/07/owl#"
        xmlns="http://www.ttmalonne.webdynamit.net/ontologies/scenario.owl#"
        xml:base="http://www.ttmalonne.webdynamit.net/ontologies/scenario.owl">

<owl:Class rdf:ID="Scenario">
        <rdfs:subClassOf>
          <owl:Restriction>
            <owl:onProperty>
                <owl:ObjectProperty rdf:ID="hasFirstOperation"/>
            </owl:onProperty>
            <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
          </owl:Restriction>
        </rdfs:subClassOf>
</owl:Class>


<owl:Class rdf:ID="ScenarioOperation">
        <rdfs:subClassOf>
          <owl:Restriction>
            <owl:onProperty>
              <owl:ObjectProperty rdf:ID="aboutOperation"/>
            </owl:onProperty>
            <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
          </owl:Restriction>
        </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="NextGrp">
        <rdfs:subClassOf>
          <owl:Restriction>
            <owl:onProperty>
              <owl:ObjectProperty rdf:ID="hasCondition"/>
            </owl:onProperty>
            <owl:maxCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:maxCardinality>
          </owl:Restriction>
```

```
        </rdfs:subClassOf>
        <rdfs:subClassOf>
          <owl:Restriction>
            <owl:onProperty>
              <owl:ObjectProperty rdf:ID="hasNextOp"/>
                </owl:onProperty>
              <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:minCardinality>
          </owl:Restriction>
        </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="NotifOperation">
        <rdfs:subClassOf rdf:resource="#ScenarioOperation"/>
        <rdfs:subClassOf>
          <owl:Restriction>
            <owl:onProperty>
              <owl:DatatypeProperty rdf:ID="hasNotifId"/>
            </owl:onProperty>
            <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
          </owl:Restriction>
        </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="AutoOperation">
        <rdfs:subClassOf rdf:resource="#ScenarioOperation"/>
        <rdfs:subClassOf>
          <owl:Restriction>
            <owl:onProperty>
              <owl:DatatypeProperty rdf:ID="hasRunLevel"/>
            </owl:onProperty>
            <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
          </owl:Restriction>
        </rdfs:subClassOf>
</owl:Class>

<owl:ObjectProperty rdf:about="#aboutOperation">
        <rdfs:range rdf:resource="http://www.ttmalonne.webdynamit.net/ontologies/services.owl#Operation"/>
        <rdfs:domain rdf:resource="#ScenarioOperation"/>
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:about="#hasRunLevel">
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
        <rdfs:domain rdf:resource="#AutoOperation"/>
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#hasNotifId">
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
        <rdfs:domain rdf:resource="#NotifOperation"/>
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#hasCondition">
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
        <rdfs:domain rdf:resource="#NextGrp"/>
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:about="#hasNext">
        <rdfs:range rdf:resource="#NextGrp"/>
        <rdfs:domain rdf:resource="#ScenarioOperation"/>
```

```
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasFirstOperation">
        <rdfs:range rdf:resource="#ScenarioOperation"/>
        <rdfs:domain rdf:resource="#Scenario"/>
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasNextOp">
        <rdfs:range rdf:resource="#ScenarioOperation"/>
        <rdfs:domain rdf:resource="#NextGrp"/>
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:about="#isEnding">
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
        <rdfs:domain rdf:resource="#ScenarioOperation"/>
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
</owl:DatatypeProperty>


</rdf:RDF>
```

Annexe B

Description OWL d'un exemple de service

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:s="http://www.ttmalonne.webdynamit.net/ontologies/services.owl#"
    xmlns:sc="http://www.ttmalonne.webdynamit.net/ontologies/scenario.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns="http://www.ttmalonne.webdynamit.net/ontologies/Film.owl#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://www.ttmalonne.webdynamit.net/ontologies/Film.owl">
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://www.ttmalonne.webdynamit.net/ontologies/services.owl"/>
    <owl:imports rdf:resource="http://www.ttmalonne.webdynamit.net/ontologies/scenario.owl"/>
  </owl:Ontology>

  <owl:Class rdf:ID="ShortText"/>
  <owl:DatatypeProperty rdf:ID="text">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
    <rdfs:domain rdf:resource="#ShortText"/>
  </owl:DatatypeProperty>

  <owl:Class rdf:ID="NumeroCarte"/>
  <owl:DatatypeProperty rdf:ID="numcarte">
          <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#integer"/>
          <rdf:type rdf:resource="http://www.w3.org.2002/07/owl#FunctionalProperty"/>
          <rdfs:domain rdf:resource="#NumeroCarte"/>
  </owl:DatatypeProperty>


  <s:ProcessInput rdf:ID="TitreFilmAchat">
    <s:hasDefaultValue>
      <s:DataInstance rdf:ID="TitreFilmAchat_DefaultValue">
        <rdf:type rdf:resource="#ShortText"/>
        <s:isInstanceOf rdf:resource="#TitreFilmAchat"/>
        <text rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Dernier Film Sorti</text>
      </s:DataInstance>
    </s:hasDefaultValue>
    <s:hasDataType rdf:resource="#ShortText"/>
    <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Choix du film (achat)</s:hasName>
  </s:ProcessInput>
```

```
<s:OperationInput rdf:ID="NomUtilisateur">
  <s:hasDataType rdf:resource="#ShortText"/>
  <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Votre nom</s:hasName>
</s:OperationInput>

<s:OperationInput rdf:ID="TitreFilmLocation">
          <s:hasDefaultValue>
            <s:DataInstance rdf:ID="TitreFilmLocation_DefaultValue">
              <rdf:type rdf:resource="#ShortText"/>
              <s:isInstanceOf rdf:resource="#TitreFilmLocation"/>
              <text rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
                            Dernier Film Sorti</text>
            </s:DataInstance>
          </s:hasDefaultValue>
          <s:hasDataType rdf:resource="#ShortText"/>
          <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
              Choix du film (location)</s:hasName>
</s:OperationInput>

<s:OperationInput rdf:ID="PayementCarteNumero">
        <s:hasDataType rdf:resource="#NumeroCarte"/>
        <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
                      Numero de votre carte</s:hasName>
</s:OperationInput>

<s:OperationInput rdf:ID="PayementFactureAdresse">
        <s:hasDataType rdf:resource="#ShortText"/>
        <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
                      Votre Adresse</s:hasName>
</s:OperationInput>

<s:OperationOutput rdf:ID="ResultatIdentification">
  <s:hasDataType rdf:resource="#ShortText"/>
  <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Statut</s:hasName>
</s:OperationOutput>

<s:OperationOutput rdf:ID="StatutPayementCarte">
          <s:hasDataType rdf:resource="#ShortText"/>
          <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
                      Payement par carte</s:hasName>
</s:OperationOutput>

<s:OperationOutput rdf:ID="StatutPayementFacture">
        <s:hasDataType rdf:resource="#ShortText"/>
        <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
                      Payement par facture</s:hasName>
</s:OperationOutput>


<s:Operation rdf:ID="Connexion">
        <s:producesOperationOutputs rdf:resource="#ResultatIdentification"/>
  <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Identification</s:hasName>
  <s:requiresOperationInputs rdf:resource="#NomUtilisateur"/>
</s:Operation>

<s:Operation rdf:ID="Location">
      <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Location</s:hasName>
      <s:requiresOperationInputs rdf:resource="#TitreFilmLocation"/>
</s:Operation>

<s:Operation rdf:ID="Achat">
```

```
        <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Achat</s:hasName>
        <s:requiresOperationInputs rdf:resource="#TitreFilmAchat"/>
</s:Operation>

<s:Operation rdf:ID="PayerFacture">
      <s:producesOperationOutputs rdf:resource="#StatutPayementFacture"/>
      <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
                  Payement par facture</s:hasName>
      <s:requiresOperationInputs rdf:resource="#PayementFactureAdresse"/>
</s:Operation>

<s:Operation rdf:ID="PayerCarte">
        <s:producesOperationOutputs rdf:resource="#StatutPayementCarte"/>
        <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
                  Payement par carte</s:hasName>
        <s:requiresOperationInputs rdf:resource="#PayementCarteNumero"/>
</s:Operation>

<s:Operation rdf:ID="Deconnexion">
        <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Deconnexion</s:hasName>
</s:Operation>

<s:Service rdf:ID="FilmService1">
  <s:hasServiceType>
    <s:ServiceType rdf:ID="FilmServiceType">
      <s:hasGui>
        <s:GUIDescription rdf:ID="FilmGUI"/>
      </s:hasGui>
      <s:isInstanciedBy rdf:resource="#FilmService1"/>
      <s:realize>
        <s:Process rdf:ID="FilmProcess">
          <s:hasTextualDescription rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
                  Service achat et location de films</s:hasTextualDescription>
          <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">HomeCinema</s:hasName>
          <s:isRealizedBy rdf:resource="#FilServiceType"/>
          <s:isDocumentedBy rdf:resource="http://www.homecinema.be"/>
        </s:Process>
      </s:realize>
      <s:hasScenario rdf:resource="#HomeCinema_Scenario"/>
      <s:presents>
        <s:Interface rdf:ID="FilmInterface">
          <s:definesOperations rdf:resource="#Connexion"/>
          <s:definesOperations rdf:resource="#Achat"/>
          <s:definesOperations rdf:resource="#Location"/>
          <s:definesOperations rdf:resource="#PayerFacture"/>
          <s:definesOperations rdf:resource="#PayerCarte"/>
          <s:definesOperations rdf:resource="#Deconnexion"/>
        </s:Interface>
      </s:presents>
    </s:ServiceType>
  </s:hasServiceType>
  <s:isCharacterizedBy>
        <s:PropertyInstance rdf:ID="FilmVersionPropertyInstance">
                  <s:isInstanceOf rdf:resource="#FilmProperty"/>
                  <rdf:type rdf:resource="#ShortText"/>
                  <text rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Home Cinema v0.1</text>
        </s:PropertyInstance>
  </s:isCharacterizedBy>
</s:Service>

<s:Property rdf:ID="FilmProperty">
        <s:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Version</s:hasName>
```

```
        <s:hasDataType rdf:resource="#ShortText"/>
    </s:Property>


    <sc:ScenarioOperation rdf:ID="op_deconnexion">
          <sc:aboutOperation rdf:resource="#Deconnexion"/>
          <sc:isEnding>true</sc:isEnding>
    </sc:ScenarioOperation>

    <sc:ScenarioOperation rdf:ID="op_location">
          <sc:aboutOperation rdf:resource="#Location"/>
          <sc:hasNext rdf:resource="#op_next3"/>
    </sc:ScenarioOperation>

    <sc:ScenarioOperation rdf:ID="op_achat">
          <sc:aboutOperation rdf:resource="#Achat"/>
          <sc:hasNext rdf:resource="#op_next3"/>
    </sc:ScenarioOperation>

    <sc:ScenarioOperation rdf:ID="op_carte">
          <sc:aboutOperation rdf:resource="#PayerCarte"/>
          <sc:isEnding>true</sc:isEnding>
    </sc:ScenarioOperation>

    <sc:ScenarioOperation rdf:ID="op_facture">
          <sc:aboutOperation rdf:resource="#PayerFacture"/>
          <sc:isEnding>true</sc:isEnding>
    </sc:ScenarioOperation>

    <sc:NextGrp rdf:ID="op_next1">
          <sc:hasNextOp rdf:resource="#op_deconnexion"/>
    </sc:NextGrp>

    <sc:NextGrp rdf:ID="op_next2">
          <sc:hasNextOp rdf:resource="#op_location"/>
         <sc:hasNextOp rdf:resource="#op_achat"/>
    </sc:NextGrp>

    <sc:NextGrp rdf:ID="op_next3">
          <sc:hasNextOp rdf:resource="#op_carte"/>
          <sc:hasNextOp rdf:resource="#op_facture"/>
    </sc:NextGrp>

    <sc:Scenario rdf:ID="HomeCinema_Scenario">
      <sc:hasFirstOperation>
                  <sc:ScenarioOperation rdf:ID="op_connex">
                          <sc:aboutOperation rdf:resource="#Connexion"/>
                          <sc:hasNext rdf:resource="#op_next1"/>
                          <sc:hasNext rdf:resource="#op_next2"/>
                  </sc:ScenarioOperation>
      </sc:hasFirstOperation>
    </sc:Scenario>

</rdf:RDF>
```